

IDC DOCUMENTATION

# Detection and Feature Extraction (DFX)

Scheme Files



**Notice**

This document was published June 2001 by the Monitoring Systems Operation of Science Applications International Corporation (SAIC) as part of the International Data Centre (IDC) Documentation. Every effort was made to ensure that the information in this document was accurate at the time of publication. However, information is subject to change.

**Contributors**

Charles N. Katz, Science Applications International Corporation  
Jeffrey A. Hanson, Science Applications International Corporation  
Ethan D. Brown, Science Applications International Corporation

**Trademarks**

BEA TUXEDO is a registered trademark of BEA Systems, Inc.  
Enterprise is a registered trademark of Sun Microsystems.  
ORACLE is a registered trademark of Oracle Corporation.  
SAIC is a trademark of Science Applications International Corporation.  
Scheme In One Defun (SIOD) is a product of Paradigm Associates of Cambridge, Massachusetts.  
Solaris is a registered trademark of Sun Microsystems.  
SPARC is a registered trademark of Sun Microsystems.  
SQL\*Plus is a registered trademark of Oracle Corporation.  
Sun is a registered trademark of Sun Microsystems.  
UltraSPARC is a registered trademark of Sun Microsystems.  
UNIX is a registered trademark of UNIX System Labs, Inc.

**Ordering Information**

The ordering number for this document is SAIC-01/3000.

This document is cited within other IDC documents as [IDC7.1.1].

# Detection and Feature Extraction (DFX) - Scheme Files

## CONTENTS

<b>About this Document</b>	i
■ PURPOSE	ii
■ SCOPE	ii
■ AUDIENCE	ii
■ RELATED INFORMATION	iii
■ USING THIS DOCUMENT	iii
Conventions	v
<b>Chapter 1: Overview</b>	1
■ INTRODUCTION	2
■ FUNCTIONALITY	8
■ IDENTIFICATION	8
■ STATUS OF DEVELOPMENT	12
■ BACKGROUND AND HISTORY	12
■ OPERATING ENVIRONMENT	13
Hardware	13
Commercial-Off-The-Shelf Software	13
<b>Chapter 2: Architectural Design</b>	15
■ CONCEPTUAL DESIGN	16
DFX Data Representations	18
DFX Functions	22
Structural Design of Scheme Scripts	23
Scheme Application Invocation	24
■ DESIGN DECISIONS	25
Programming Language	26
Global Libraries	26

Database	27
Interprocess Communication (IPC)	27
Filesystem	27
Design Model	28
Database Schema Overview	28
■ <b>FUNCTIONAL DESCRIPTION</b>	30
Shared Functionality	30
Seismic Processing	32
Hydroacoustic Processing	34
Infrasonic Processing	36
Administrative Processing	36
■ <b>INTERFACE DESIGN</b>	37
Interface with Other IDC Systems	37
Interface with External Users	38
Interface with Operators	38
<b>Chapter 3: Shared Functionality</b>	39
■ <b>DATABASE OPERATIONS</b>	40
Opening and Closing the Database	41
Querying the Database	41
Screening Database Containers	43
Writing to the Database	44
■ <b>INITIALIZING SITES</b>	46
Contents of Initsite Objects	46
Creating Initsite Objects	52
■ <b>READING WAVEFORMS</b>	58
Waveform Objects	58
Determining Data Time Intervals	64
Reading Binary Files	69
Checking Quality of Waveforms	69
■ <b>COMMON DFX SCHEME TASKS</b>	71
Defining Generic Objects	71
Looping Over Contents of Containers	74

Internal Screening of Containers	74
Setting and Retrieving Parameters from CVAR Datastore	75
Error Handling	76
Memory Management	77
<b>Chapter 4: Seismic Processing</b>	<b>79</b>
■ <b>AUTOMATIC SEISMIC DETECTION [1]</b>	<b>80</b>
General Description	81
Architecture	81
Input/Output	83
Processes	88
Detailed Description of Generate and Write TM Beams [1.3]	101
Detailed Description of Generate Pre-existing Arrival Beams [1.4]	108
Detailed Description of Perform Detection Processing [1.5]	113
Detailed Description of Revise Extracted Estimates [1.6]	122
Detailed Description of Perform Feature Extraction [1.8]	132
■ <b>AUTOMATIC ORIGIN BEAM [2]</b>	<b>148</b>
General Description	148
Architecture	149
Input/Output	151
Processes	153
Detailed Description of Generate and Write Origin Beams [2.3]	156
■ <b>INTERACTIVE BEAM-ON-THE-FLY [3]</b>	<b>164</b>
General Description	164
Architecture	165
Input/Output	166
Processes	168
Detailed Description of Initialize Sites [3.2]	170
Detailed Description of Generate and Write Origin Beam [3.3]	178
■ <b>INTERACTIVE SEISMIC RECALL [4]</b>	<b>183</b>
General Description	183
Architecture	184
Input/Output	184
Processes	189

Detailed Description of Initialize Travel-time Tables [4.2]	192
Detailed Description of Perform Recall Signal Processing [4.4]	194
■ <b>AUTOMATIC SEISMIC RECALL [5]</b>	202
General Description	202
Architecture	203
Input/Output	203
Processes	208
Detailed Description of Perform Recall Signal Processing [5.4]	213
■ <b>AUTOMATIC DEPTH-PHASE SNR [6]</b>	218
General Description	218
Architecture	219
Input/Output	219
Processes	223
Detailed Description of Estimate Depth-phase SNR [6.3]	226
■ <b>AUTOMATIC NOISE AMPLITUDE ESTIMATION [7]</b>	232
General Description	232
Architecture	233
Input/Output	235
Processes	237
Detailed Description of Estimate Noise Amplitude [7.5]	240
■ <b>AUTOMATIC SEISMIC EVENT CHARACTERIZATION [8]</b>	249
General Description	249
Architecture	250
Input/Output	252
Processes	259
Detailed Description of Estimate Event Characteristics [8.3]	266
<b>Chapter 5: Hydroacoustic Processing</b>	289
■ <b>AUTOMATIC HYDROACOUSTIC DETECTION [9]</b>	290
General Description	291
Architecture	291
Input/Output	292
Processes	295

Detailed Description of Make Detections [9.3]	304
Detailed Description of Measure Hydroacoustic Detection Features [9.6]	307
■ <b>INTERACTIVE HYDROACOUSTIC RECALL [10]</b>	312
General Description	312
Architecture	313
Input/Output	313
Processes	317
Detailed Description of Perform Recall Signal Processing [10.4]	319
■ <b>AUTOMATIC HYDROACOUSTIC EVENT CHARACTERIZATION [11]</b>	323
General Description	323
Architecture	324
Input/Output	325
Processes	328
Detailed Description of Estimate Event Characteristics [11.3]	330
<b>Chapter 6: Infrasonic Processing</b>	339
■ <b>AUTOMATIC INFRASONIC DETECTION [12]</b>	340
General Description	341
Architecture	341
Input/Output	342
Processes	347
Detailed Description of Generate Primary Detection Space (PDS) [12.3]	366
Detailed Description of Generate Coherence Traces as Beams [12.4]	376
Detailed Description of Remove Redundant Detections [12.6]	381
Detailed Description of Estimate Signal Amplitude [12.7]	391
Detailed Description of Write Non-Redundant Detections [12.8]	399
<b>Chapter 7: Administrative Processing</b>	405
■ <b>SEGMENT ARCHIVING [13]</b>	406
General Description	407
Architecture	407
Input/Output	409
Processes	412
Detailed Description of Perform Segment Archiving [13.3]	415

■ QUALITY CONTROL STATISTICS [14]	423
General Description	423
Architecture	423
Input/Output	424
Processes	427
Detailed Description of Identify Missing Waveforms [14.1]	432
Detailed Description of Calculate Waveforms Availability [14.3]	437
<b>Chapter 8: Database Description</b>	441
■ GENERAL DESCRIPTION	442
■ DATABASE TABLES	442
■ OUTPUT DATABASE ATTRIBUTES	444
Seismic Processing	444
Hydroacoustic Processing	449
Infrasonic Processing	451
Administrative Processing	451
<b>References</b>	453
<b>Glossary</b>	G1
<b>Index</b>	I1



# Detection and Feature Extraction (DFX) - Scheme Files

## FIGURES

FIGURE 1.	IDC SOFTWARE CONFIGURATION HIERARCHY	3
FIGURE 2.	RELATIONSHIP OF DFX TO OTHER SOFTWARE UNITS IN AUTOMATIC PROCESSING	5
FIGURE 3.	RELATIONSHIP OF DFX TO OTHER SOFTWARE UNITS IN INTERACTIVE PROCESSING	7
FIGURE 4.	RELATIONSHIP OF DFX TO OTHER SOFTWARE UNITS IN AUTOMATIC POST-ANALYSIS PROCESSING	9
FIGURE 5.	DFX EXTERNAL DATA INTERFACE	17
FIGURE 6.	DFX EXTERNAL DATA SOURCES AND INTERNAL DATA REPRESENTATIONS	19
FIGURE 7.	ARCHITECTURE OF CREATE-INITSITE	53
FIGURE 8.	ARCHITECTURE OF INITIALIZE-NET-SITES	55
FIGURE 9.	ARCHITECTURE OF CREATE-INITSITE-CONTAINER-FROM-DBSITE-CONTAINER	56
FIGURE 10.	ARCHITECTURE OF FILL-INITSITE-CONTAINER	57
FIGURE 11.	COMPUTED DATA AND PROCESSING TIME INTERVALS	68
FIGURE 12.	ARCHITECTURE OF SEISDET	82
FIGURE 13.	ARCHITECTURE OF GENERATE AND WRITE TM BEAMS	102
FIGURE 14.	ARCHITECTURE OF GENERATE PRE-EXISTING ARRIVAL BEAMS	109
FIGURE 15.	ARCHITECTURE OF PERFORM DETECTION PROCESSING	114
FIGURE 16.	ARCHITECTURE OF REVISE EXTRACTED ESTIMATES	123
FIGURE 17.	ARCHITECTURE OF PERFORM FEATURE EXTRACTION	133
FIGURE 18.	ARCHITECTURE OF ORIGINBEAM	150
FIGURE 19.	ARCHITECTURE OF GENERATE AND WRITE ORIGIN BEAMS	157
FIGURE 20.	ARCHITECTURE OF BOTF	165
FIGURE 21.	ARCHITECTURE OF INITIALIZE SITES	172
FIGURE 22.	ARCHITECTURE OF GENERATE BEAM	179
FIGURE 23.	ARCHITECTURE OF INTSEISRCL	185

FIGURE 24.	ARCHITECTURE OF INITIALIZE TRAVEL TIMES	193
FIGURE 25.	ARCHITECTURE OF PERFORM RECALL SIGNAL PROCESSING	195
FIGURE 26.	ARCHITECTURE OF AUTOSeisRCL	204
FIGURE 27.	ARCHITECTURE OF PERFORM RECALL SIGNAL PROCESSING	214
FIGURE 28.	ARCHITECTURE OF DPSNR	220
FIGURE 29.	ARCHITECTURE OF PERFORM ESTIMATE DEPTH-PHASE SNR	227
FIGURE 30.	ARCHITECTURE OF NOISEAMP	234
FIGURE 31.	ARCHITECTURE OF ESTIMATE NOISE AMPLITUDE	241
FIGURE 32.	ARCHITECTURE OF SeisEVCH	251
FIGURE 33.	ARCHITECTURE OF ESTIMATE EVENT CHARACTERISTICS	267
FIGURE 34.	ARCHITECTURE OF HYDRODET	292
FIGURE 35.	ARCHITECTURE OF MAKE DETECTIONS	304
FIGURE 36.	ARCHITECTURE OF MEASURE HYDROACOUSTIC DETECTION FEATURES	308
FIGURE 37.	ARCHITECTURE OF INTHYDROCL	314
FIGURE 38.	ARCHITECTURE OF PERFORM RECALL SIGNAL PROCESSING	320
FIGURE 39.	ARCHITECTURE OF HYDROEVCH	325
FIGURE 40.	ARCHITECTURE OF ESTIMATE EVENT CHARACTERISTICS	332
FIGURE 41.	ARCHITECTURE OF INFRADET	343
FIGURE 42.	ARCHITECTURE OF GENERATE PRIMARY DETECTION SPACE (PDS)	367
FIGURE 43.	ARCHITECTURE OF GENERATE COHERENCE TRACES AS BEAMS	377
FIGURE 44.	ARCHITECTURE OF REMOVE REDUNDANT DETECTIONS	382
FIGURE 45.	ARCHITECTURE OF ESTIMATE SIGNAL AMPLITUDE	392
FIGURE 46.	ARCHITECTURE OF WRITE NON-REDUNDANT DETECTIONS	401
FIGURE 47.	ARCHITECTURE OF SEGARCH	408
FIGURE 48.	ARCHITECTURE OF PERFORM SEGMENT ARCHIVING	416
FIGURE 49.	ARCHITECTURE OF QCSTATS	424
FIGURE 50.	ARCHITECTURE OF IDENTIFY MISSING WAVEFORMS	433
FIGURE 51.	ARCHITECTURE OF CALCULATE WAVEFORMS AVAILABILITY	438

# Detection and Feature Extraction (DFX) - Scheme Files

## TABLES

TABLE I:	DATA FLOW SYMBOLS	vi
TABLE II:	TYPOGRAPHICAL CONVENTIONS	vii
TABLE 1:	DFX APPLICATIONS	10
TABLE 2:	ABBREVIATIONS AND SYSTEM NAMES FOR DFX APPLICATIONS	11
TABLE 3:	DATABASE TABLES USED BY DFX	28
TABLE 4:	PARAMETERS STORED IN INITSITE	47
TABLE 5:	DATABASE OBJECTS STORED IN INITSITE	47
TABLE 6:	RECIPES STORED IN INITSITE	48
TABLE 7:	WAVEFORM OBJECTS STORED IN INITSITE OBJECTS	49
TABLE 8:	OBJECTS IN INITSITE PER APPLICATION	50
TABLE 9:	METHOD USED TO CREATE INITSITE	52
TABLE 10:	FUNCTIONAL INTERFACE OF QUERY-FOR-WFDATA-BY-STA-CHAN-TIME	58
TABLE 11:	CONTENTS OF WFDATA OBJECT	59
TABLE 12:	FUNCTIONAL INTERFACE OF QUERY-FOR-WFMEM-BY-STA-CHAN-TIME	62
TABLE 13:	CONTENTS OF WFMEM OBJECT	63
TABLE 14:	FUNCTIONAL INTERFACE OF COMPUTE-SITE-ORIGIN-TI	64
TABLE 15:	FUNCTIONAL INTERFACE OF GET-DATA-INTERVAL	66
TABLE 16:	FUNCTIONAL INTERFACE OF GET-PROCESSING-INTERVAL	66
TABLE 17:	FUNCTIONAL INTERFACE OF READ-WAVEFORMS	69
TABLE 18:	FUNCTIONAL INTERFACE OF QC-WAVEFORMS AND QC-WAVEFORMS-ALL	70
TABLE 19:	COMMON GENERIC OBJECT TYPES	72
TABLE 20:	INPUT PARAMETERS FOR AUTOMATIC SEISMIC DETECTION PROCESSING	84
TABLE 21:	DATA PRODUCED BY AUTOMATIC SEISMIC DETECTION PROCESSING	87
TABLE 22:	FUNCTIONAL INTERFACE OF CREATE-DETECTION-INITSITE	88
TABLE 23:	INTERNAL SYMBOLS FOR INITIALIZE SITES	89

TABLE 24:	FUNCTIONAL INTERFACE OF GET-DETECTION-CHANNEL-LIST	90
TABLE 25:	FUNCTIONAL INTERFACE OF READ-DETECTION-WAVEFORMS	91
TABLE 26:	FUNCTIONAL INTERFACE OF MAKE-AND-WRITE-TM-BEAMS	93
TABLE 27:	FUNCTIONAL INTERFACE OF MAKE-PRE-EXISTING-ARRIVAL-BEAMS	95
TABLE 28:	FUNCTIONAL INTERFACE OF MAKE-DETECTIONS	96
TABLE 29:	FUNCTIONAL INTERFACE OF REVISE-DETECTION-ESTIMATES	97
TABLE 30:	FUNCTIONAL INTERFACE OF SCREEN-DETECTION-CONTAINER	99
TABLE 31:	FUNCTIONAL INTERFACE OF MEASURE-DETECTION-FEATURES	100
TABLE 32:	PARAMETERS ACQUIRED BY EXTRACT PARAMETERS AND SAVE AND RESET STAV-LEN	103
TABLE 33:	FUNCTIONAL INTERFACE OF MAKE-BEAM-FROM-DATA	104
TABLE 34:	FUNCTIONAL INTERFACE OF MAKE-NUL-Beam	105
TABLE 35:	FUNCTIONAL INTERFACE OF COMPUTE-SNR	106
TABLE 36:	FUNCTIONAL INTERFACE OF WRITE-TM-STA	107
TABLE 37:	PARAMETERS ACQUIRED BY EXTRACT PARAMETERS	109
TABLE 38:	FUNCTIONAL INTERFACE OF QUERY-FOR-DBARRIVALS-WITH-BEAMS	110
TABLE 39:	PARAMETERS SET IN LOCAL DETECTION GOBJ	111
TABLE 40:	FUNCTIONAL INTERFACE OF MAKE-DET-BEAMS	112
TABLE 41:	PARAMETERS ACQUIRED BY EXTRACT PARAMETERS	115
TABLE 42:	FUNCTIONAL INTERFACE OF MAKE-BEAM-FROM-DATA	116
TABLE 43:	FUNCTIONAL INTERFACE OF COMPUTE-SNR	118
TABLE 44:	FUNCTIONAL INTERFACE OF FIND-TRIGGERS	119
TABLE 45:	FUNCTIONAL INTERFACE OF FIND-DETECTIONS	120
TABLE 46:	PARAMETERS ACQUIRED BY EXTRACT PARAMETERS	124
TABLE 47:	FUNCTIONAL INTERFACE OF REVISE-DET-ONSET	125
TABLE 48:	FUNCTIONAL INTERFACE OF COMPUTE-DET-AMP-BEST-BEAM	127
TABLE 49:	FUNCTIONAL INTERFACE OF MAKE-FK	128
TABLE 50:	FUNCTIONAL INTERFACE OF MEASURE-DETECTION-POLARIZATION	131
TABLE 51:	PARAMETERS ACQUIRED BY EXTRACT PARAMETERS	134
TABLE 52:	FUNCTIONAL INTERFACE OF CREATE-DBARRIVAL-FROM-DETECTION	136

TABLE 53:	FUNCTIONAL INTERFACE OF SUBMIT-OBJECT-DB (ARRIVAL)	136
TABLE 54:	FUNCTIONAL INTERFACE OF CREATE-DBAPMA-FROM-DETECTION	137
TABLE 55:	FUNCTIONAL INTERFACE OF SUBMIT-OBJECT-DB (APMA)	138
TABLE 56:	FUNCTIONAL INTERFACE OF MAKE-DET-AMPLITUDES	139
TABLE 57:	FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (AMPLITUDE)	140
TABLE 58:	FUNCTIONAL INTERFACE OF MAKE-DET-BEAMS	141
TABLE 59:	FUNCTIONAL INTERFACE OF CREATE-DBDETECTION-FROM-DETECTION	142
TABLE 60:	FUNCTIONAL INTERFACE OF SUBMIT-OBJECT-DB (DETECTION)	143
TABLE 61:	FUNCTIONAL INTERFACE OF CREATE-SBSNR-DBAMPLITUDE-FROM-DET	144
TABLE 62:	FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (SBSNR)	145
TABLE 63:	FUNCTIONAL INTERFACE OF CREATE-DBAMP3C-FROM-DETECTION	146
TABLE 64:	FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (AMP3C)	147
TABLE 65:	INPUT PARAMETERS FOR AUTOMATIC ORIGIN BEAM PROCESSING	151
TABLE 66:	DATA PRODUCED BY AUTOMATIC ORIGIN BEAM PROCESSING	153
TABLE 67:	FUNCTIONAL INTERFACE OF QUERY-FOR-DBORIGIN-BY-TIME	154
TABLE 68:	FUNCTIONAL INTERFACE OF INITIALIZE-NET-SITES	155
TABLE 69:	FUNCTIONAL INTERFACE OF PROCESS-ORIGIN-FOR-INITSITE	156
TABLE 70:	PARAMETERS EXTRACTED FROM INITSITE AND DBORIGIN OBJECTS BY PROCESS-ORIGIN-FOR-INITSITE	158
TABLE 71:	INTERNAL PARAMETERS FOR PROCESS-ORIGIN-FOR-INITSITE	159
TABLE 72:	FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-INITSITE	160
TABLE 73:	FUNCTIONAL INTERFACE OF COMPUTE-ORIGIN-BEAMS	161
TABLE 74:	FUNCTIONAL INTERFACE OF SUBMIT-AUX	163
TABLE 75:	INPUT PARAMETERS FOR INTERACTIVE BEAM-ON-THE-FLY PROCESSING	166
TABLE 76:	DATA PRODUCED BY INTERACTIVE BEAM-ON-THE-FLY PROCESSING	168
TABLE 77:	FUNCTIONAL INTERFACE OF INITIALIZE-STATION-SITES	169
TABLE 78:	FUNCTIONAL INTERFACE OF CREATE-STATION-EVENT-BEAM	170
TABLE 79:	FUNCTIONAL INTERFACE OF QUERY-FOR-STATIONS-SITE-CONTAINER	173
TABLE 80:	FUNCTIONAL INTERFACE OF INITIALIZE-SITES	174
TABLE 81:	FUNCTIONAL INTERFACE OF READ-DEFAULT-TRAVEL-TIME-TABLES	174

TABLE 82:	FUNCTIONAL INTERFACE OF COMPUTE-STATIONS-TI	175
TABLE 83:	FUNCTIONAL INTERFACE OF QUERY-FOR-WFDATA-CONTAINER	176
TABLE 84:	FUNCTIONAL INTERFACE OF QUERY-FOR-WFMEM-CONTAINER	177
TABLE 85:	FUNCTIONAL INTERFACE OF COMPUTE-ORIGIN-BEAMS	181
TABLE 86:	FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (WFDISC)	182
TABLE 87:	INPUT PARAMETERS FOR INTERACTIVE SEISMIC RECALL PROCESSING	186
TABLE 88:	DATA WRITTEN BY INTERACTIVE SEISMIC RECALL PROCESSING	188
TABLE 89:	FUNCTIONAL INTERFACE OF INITIALIZE-FOR-TRAVEL-TIME	190
TABLE 90:	FUNCTIONAL INTERFACE OF CREATE-RECALL-INITSITE	191
TABLE 91:	FUNCTIONAL INTERFACE OF PROCESS-RECALL-ARRIVAL	192
TABLE 92:	FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-RECALL	196
TABLE 93:	FUNCTIONAL INTERFACE OF FIND-RECALL-BEST-BEAM	197
TABLE 94:	FUNCTIONAL INTERFACE OF REVISE-RECALL-FEATURES	198
TABLE 95:	FUNCTIONS CALLED WITHIN REVISE-RECALL-FEATURES	198
TABLE 96:	DATABASE OBJECTS DEFINED IN MEASURE-RECALL-FEATURES	200
TABLE 97:	FUNCTIONAL INTERFACE OF MEASURE-RECALL-FEATURES	201
TABLE 98:	INPUT PARAMETERS FOR AUTOMATIC SEISMIC RECALL PROCESSING	205
TABLE 99:	DATA UPDATED BY AUTOMATIC SEISMIC RECALL PROCESSING	207
TABLE 100:	INTERNAL SYMBOLS FOR QUERY-FOR-DBARRIVAL-CONTAINER	209
TABLE 101:	POSSIBLE ARRIVAL QUERIES	209
TABLE 102:	FUNCTIONAL INTERFACE OF INITIALIZE-FOR-RESIDUAL-UPDATE	210
TABLE 103:	DATABASE QUERY FUNCTIONS USED BY INITIALIZE-FOR-RESIDUAL-UPDATE	211
TABLE 104:	FUNCTIONAL INTERFACE OF CREATE-RECALL-INITSITE	212
TABLE 105:	FUNCTIONAL INTERFACE OF PROCESS-RECALL-ARRIVAL	213
TABLE 106:	FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-RECALL	215
TABLE 107:	FUNCTIONAL INTERFACE OF SCREEN-RECALL-DETECTIONS	217
TABLE 108:	INPUT PARAMETERS FOR DEPTH-PHASE SNR PROCESSING	221
TABLE 109:	DATA PRODUCED BY DEPTH-PHASE SNR PROCESSING	223
TABLE 110:	INTERNAL SYMBOLS FOR QUERY-FOR-DBARRIVAL-DEPTH-PHASES	224
TABLE 111:	FUNCTIONAL INTERFACE OF CREATE-DEPTH-PHASE-INITSITE	225

TABLE 112:	FUNCTIONAL INTERFACE OF PROCESS-DEPTH-PHASE-ARRIVAL	225
TABLE 113:	FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-DEPTH-PHASE	228
TABLE 114:	FUNCTIONAL INTERFACE OF MEASURE-DETECTION-AMPLITUDE	229
TABLE 115:	INPUT PARAMETERS FOR AUTOMATIC NOISE AMPLITUDE ESTIMATION	235
TABLE 116:	DATA PRODUCED BY AUTOMATIC NOISE AMPLITUDE ESTIMATION	237
TABLE 117:	INTERNAL SYMBOLS FOR PROCESS-ORIGINS	239
TABLE 118:	FUNCTIONAL INTERFACE OF PROCESS-ORIGIN-FOR-INITSITE	239
TABLE 119:	INTERNAL SYMBOLS FOR EXTRACT PARAMETERS	242
TABLE 120:	FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-INITSITE	244
TABLE 121:	FUNCTIONAL INTERFACE OF MEASURE-ORIGIN-AMPLITUDES	246
TABLE 122:	FUNCTIONAL INTERFACE OF GC-WFDATA-CON	248
TABLE 123:	INPUT PARAMETERS FOR SEISMIC EVENT CHARACTERIZATION PROCESSING	252
TABLE 124:	DATA PRODUCED BY SEISMIC EVENT CHARACTERIZATION PROCESSING	259
TABLE 125:	FUNCTIONAL INTERFACE OF QUERY-FOR-DBORIGIN-BY-TIME	260
TABLE 126:	INTERNAL SYMBOLS FOR INITIALIZE-NET-SITES	261
TABLE 127:	INTERNAL SYMBOLS SET FROM DATABASE QUERIES WITHIN INITIALIZE-NET-SITES	263
TABLE 128:	FUNCTIONAL INTERFACE OF QUERY-FOR-DBORIGIN-BY-TIME	264
TABLE 129:	CRITERIA TO BE SATISFIED FOR FEATURE ESTIMATION	268
TABLE 130:	FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-INITSITE	270
TABLE 131:	FUNCTIONAL INTERFACE OF MEASURE-ORIGIN-AMPLITUDES	272
TABLE 132:	FUNCTIONAL INTERFACE OF SCREEN-ORIGIN-DBAMPLITUDE- BY-EXISTING-DBAMPLITUDE	273
TABLE 133:	FUNCTIONAL INTERFACE OF MEASURE-ARRIVAL-AMPLITUDES FOR OBSERVED ARRIVALS	274
TABLE 134:	FUNCTIONAL INTERFACE OF SCREEN-ARRIVAL-DBAMPLITUDE- BY-EXISTING FOR OBSERVED ARRIVALS	275
TABLE 135:	FUNCTIONAL INTERFACE OF MEASURE-ARRIVAL-AMPLITUDES FOR FIRST MOTION	276
TABLE 136:	FUNCTIONAL INTERFACE OF SCREEN-ARRIVAL-DBAMPLITUDE- BY-EXISTING FOR FIRST MOTION	277
TABLE 137:	FUNCTIONAL INTERFACE OF COMPUTE-TF	278

TABLE 138:	FUNCTIONAL INTERFACE OF SCREEN-TF-BY-EXISTING-TF	279
TABLE 139:	FUNCTIONAL INTERFACE OF COMPUTE-TMF	280
TABLE 140:	FUNCTIONAL INTERFACE OF SCREEN-TMF-BY-EXISTING-TMF	281
TABLE 141:	FUNCTIONAL INTERFACE OF COMPUTE-SPLP	282
TABLE 142:	FUNCTIONAL INTERFACE OF SCREEN-SPLP-BY-EXISTING-SPLP	282
TABLE 143:	FUNCTIONAL INTERFACE OF MAKE-SMULT	283
TABLE 144:	FUNCTIONAL INTERFACE OF SCREEN-CEPPKS-BY-EXISTING-CEPPKS	284
TABLE 145:	FUNCTIONAL INTERFACE OF SCREEN-SPVAR-BY-EXISTING-SPVAR	285
TABLE 146:	FUNCTIONAL INTERFACE OF COMPUTE-COMPLEXITY	286
TABLE 147:	FUNCTIONAL INTERFACE OF SCREEN-COMPLEXITY-BY-EXISTING-COMPLEXITY	287
TABLE 148:	INPUT PARAMETERS FOR AUTOMATIC HYDROACOUSTIC DETECTION PROCESSING	293
TABLE 149:	DATA PRODUCED BY AUTOMATIC HYDROACOUSTIC DETECTION PROCESSING	295
TABLE 150:	FUNCTIONAL INTERFACE OF CREATE-HYDRO-DETECTION-INITSITE	296
TABLE 151:	FUNCTIONAL INTERFACE OF READ-HYDRO-DETECTION-WAVEFORMS	297
TABLE 152:	FUNCTIONAL INTERFACE OF MAKE-HYDRO-DETECTIONS	298
TABLE 153:	FUNCTIONAL INTERFACE OF REVISE-HYDRO-DETECTION-ESTIMATES	299
TABLE 154:	FUNCTIONAL INTERFACE OF SCREEN-DETECTIONS-BY-TIME	300
TABLE 155:	FUNCTIONAL INTERFACE OF MEASURE-HYDRO-DETECTION-FEATURES	301
TABLE 156:	FUNCTIONAL INTERFACE OF SCREEN-HYDRO-DETECTIONS	302
TABLE 157:	FUNCTIONAL INTERFACE OF SUBMIT-DB-HYDRO-RESULTS	303
TABLE 158:	FUNCTIONAL INTERFACE OF MAKE-BEAM-FROM-DATA	306
TABLE 159:	FUNCTIONAL INTERFACE OF FIND-TRIGGERS	307
TABLE 160:	FUNCTIONAL INTERFACE OF MEASURE-HYDRO	310
TABLE 161:	FUNCTIONAL INTERFACE OF REVISE-HYDRO-DET-ATTRIBUTES	311
TABLE 162:	INPUT PARAMETERS FOR INTERACTIVE HYDROACOUSTIC RECALL PROCESSING	315
TABLE 163:	DATA PRODUCED BY INTERACTIVE HYDROACOUSTIC RECALL PROCESSING	316
TABLE 164:	FUNCTIONAL INTERFACE OF CREATE-HYDRO-RECALL-INITSITE	318
TABLE 165:	FUNCTIONAL INTERFACE OF PROCESS-RECALL-ARRIVAL	319
TABLE 166:	FUNCTIONAL INTERFACE OF MEASURE-HYDRO-RECALL-FEATURES	322



TABLE 167:	INPUT PARAMETERS FOR HYDROACOUSTIC EVENT CHARACTERIZATION PROCESSING	326
TABLE 168:	DATA PRODUCED BY HYDROACOUSTIC EVENT CHARACTERIZATION PROCESSING	328
TABLE 169:	FUNCTIONAL INTERFACE OF INITIALIZE-NET-SITES	329
TABLE 170:	FUNCTIONAL INTERFACE OF PROCESS-ORIGIN-FOR-INITSITE	330
TABLE 171:	FUNCTIONAL INTERFACE OF SITE-HYDRO-BLOCKED	333
TABLE 172:	FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-INITSITE	334
TABLE 173:	FUNCTIONAL INTERFACE OF MEASURE-ORIGIN-AMPLITUDES	335
TABLE 174:	FUNCTIONAL INTERFACE OF COMPUTE-AMPBEAM-LTA AND COMPUTE-AMPBEAM-LTA-SQUARED	336
TABLE 175:	FUNCTIONAL INTERFACE OF FIND-STAV-MEASURE-FOR-BEAM-NAME	336
TABLE 176:	FUNCTIONAL INTERFACE OF SCREEN-EXISTING-HEVCH-PARRIVALS	337
TABLE 177:	INPUT PARAMETERS FOR AUTOMATIC INFRASONIC DETECTION PROCESSING	344
TABLE 178:	DATA PRODUCED BY AUTOMATIC INFRASONIC DETECTION PROCESSING	346
TABLE 179:	RECIPE INFORMATION INITIALIZED FOR INFRADET PROCESSING	348
TABLE 180:	FUNCTIONAL INTERFACE OF INIT-AMPLITUDE-RECIPE	349
TABLE 181:	FUNCTIONAL INTERFACE OF GET-INFRA-WAVEFORMS	349
TABLE 182:	FUNCTIONAL INTERFACE OF FUNCTIONS USED TO GENERATE PRIMARY DETECTION SPACE (PDS)	354
TABLE 183:	FUNCTIONAL INTERFACE OF INFRA-BEAM-FROM-SCORE-BOARDS	357
TABLE 184:	AUXILIARY BEAMS CREATED AND WRITTEN BY AUTOMATIC INFRASONIC DETECTION	358
TABLE 185:	FUNCTIONAL INTERFACE OF FUNCTIONS USED TO GENERATE COINCIDENCE DETECTION SPACE (CDS)	359
TABLE 186:	FUNCTIONAL INTERFACE OF INSTRUMENT-RESPONSE-FILES-EXIST?	364
TABLE 187:	FUNCTIONAL INTERFACE OF MEASURE-INFRA-AMPLITUDE	365
TABLE 188:	FUNCTIONAL INTERFACE OF INFRA-FIR-FILTER-DATA	368
TABLE 189:	FUNCTIONAL INTERFACE OF INFRA-CROSS-CORRELATE-PAIRS	369
TABLE 190:	FUNCTIONAL INTERFACE OF INFRA-GENERATE-SLOWNESS-PLANES-CC	371
TABLE 191:	FUNCTIONAL INTERFACE OF INFRA-PEAK-EVAL-SLOW-PLANES	373
TABLE 192:	FUNCTIONAL INTERFACE OF INFRA-BEAM-FROM-SCORE-BOARDS	378

TABLE 193:	FUNCTIONAL INTERFACE OF WRITE-DB-BEAM	379
TABLE 194:	FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (WFDISC)	380
TABLE 195:	DBCONTAINERS CREATED BY CREATE DBCONTAINERS	383
TABLE 196:	PARAMETERS EXTRACTED BY EXTRACT PARAMETERS	384
TABLE 197:	FUNCTIONAL INTERFACE OF DELETE-REDUNDANT-DET-OBJECTS	386
TABLE 198:	FUNCTIONAL INTERFACE OF SCREEN-INFRA-DBARRIVALS-BY-EXISTING	387
TABLE 199:	FUNCTIONAL INTERFACE OF SCREEN-INFRA-DBDETECTS-BY-EXISTING	389
TABLE 200:	FUNCTIONAL INTERFACE OF SCREEN-INFRA-DBINFRA_FEATURES-BY-EXISTING	390
TABLE 201:	PARAMETERS EXTRACTED AND SYMBOLS SET FOR MEASURE-INFRA-AMPLITUDE	393
TABLE 202:	FUNCTIONAL INTERFACE OF CREATE-SECONDARY-BEAM-ELEMENT-OBJECT	395
TABLE 203:	FUNCTIONAL INTERFACE OF CREATE-SECONDARY-BEAM-RECIPE-CONTAINER	396
TABLE 204:	FUNCTIONAL INTERFACE OF MEASURE-AMPLITUDE	397
TABLE 205:	FUNCTIONAL INTERFACE OF SELECT-DBAMPLITUDE-OBJECT-TO-REPORT	399
TABLE 206:	FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB	402
TABLE 207:	FUNCTIONAL INTERFACE OF DEPART-WITH-GRACE	403
TABLE 208:	INPUT PARAMETERS FOR SEGMENT ARCHIVING PROCESSING	409
TABLE 209:	DATA PRODUCED BY SEGMENT ARCHIVING PROCESSING	411
TABLE 210:	FUNCTIONAL INTERFACE OF PROCESS-ORIGINS	414
TABLE 211:	FUNCTIONAL INTERFACE OF PROCESS-ORIGIN	414
TABLE 212:	FUNCTIONAL INTERFACE OF COMPUTE-SITE-ORIGIN-TI	417
TABLE 213:	FUNCTIONAL INTERFACE OF COMPUTE-TI-PARAMETERS	418
TABLE 214:	FUNCTIONAL INTERFACE OF READ-WAVEFORMS	419
TABLE 215:	FUNCTIONAL INTERFACE OF COMPUTE-ORIGIN-BEAMS	420
TABLE 216:	INPUT PARAMETERS FOR QUALITY CONTROL STATISTICS PROCESSING	425
TABLE 217:	DATA PRODUCED BY QUALITY CONTROL STATISTICS PROCESSING	426
TABLE 218:	FUNCTIONAL INTERFACE OF PROCESS-QCSTATS	426
TABLE 219:	FUNCTIONAL INTERFACE OF FIND-MISSING-DATA	428
TABLE 220:	INFORMATION EXTRACTED FROM EACH WFDATA OBJECT'S MASK	429
TABLE 221:	FUNCTIONAL INTERFACE OF MEASURE-DATA-AVAILABILITY	430
TABLE 222:	INFORMATION STORED IN EACH QC-STATS OBJECT	431

TABLE 223:	FUNCTIONAL INTERFACE OF WRITE-QCSTATS	432
TABLE 224:	INTERNAL PARAMETERS OF FIND-MISSING-DATA	434
TABLE 225:	ATTRIBUTES SET IN QC-STATS OBJECT	435
TABLE 226:	FUNCTIONAL INTERFACE OF WRITE-QCSTATS	436
TABLE 227:	ATTRIBUTES POPULATED IN QC-STATS OBJECT	437
TABLE 228:	INTERNAL PARAMETERS OF MEASURE-DATA-AVAILABILITY	439
TABLE 229:	DATABASE TABLES USED BY EACH DFX APPLICATION	443
TABLE 230:	DATABASE ATTRIBUTES WRITTEN BY SEISDET	445
TABLE 231:	DATABASE ATTRIBUTES WRITTEN BY ORIGINBEAM	445
TABLE 232:	DATABASE ATTRIBUTES WRITTEN BY BOTF	446
TABLE 233:	DATABASE ATTRIBUTES WRITTEN BY INTSEISRCL	446
TABLE 234:	DATABASE ATTRIBUTES WRITTEN BY AUTOSEISRCL	447
TABLE 235:	DATABASE ATTRIBUTES WRITTEN BY DPSNR	447
TABLE 236:	DATABASE ATTRIBUTES WRITTEN BY NOISEAMP	448
TABLE 237:	DATABASE ATTRIBUTES WRITTEN BY SEISEVCH	448
TABLE 238:	DATABASE ATTRIBUTES WRITTEN BY HYDRODET	449
TABLE 239:	DATABASE ATTRIBUTES WRITTEN BY INTHYDRORCL	450
TABLE 240:	DATABASE ATTRIBUTES WRITTEN BY HYDROEVCH	450
TABLE 241:	DATABASE ATTRIBUTES WRITTEN BY INFRADET	451
TABLE 242:	DATABASE ATTRIBUTES WRITTEN BY SEGARCH	452
TABLE 243:	DATABASE ATTRIBUTES WRITTEN BY QCSTATS	452



## About this Document

This chapter describes the organization and content of the document and includes the following topics:

- Purpose
- Scope
- Audience
- Related Information
- Using this Document

# About this Document

## PURPOSE

This document describes the design of applications using the *Detection and Feature Extraction (DFX)* software of the International Data Centre (IDC). The software is a computer software component (CSC) of the Automatic Processing Computer Software Configuration Item (CSCI). The *DFX* applications described in this document are designed to be part of a pipeline processing unit or to be invoked from within the *Analyst Review Station (ARS)*. This document describes the applications that are currently used in IDC operations.

## SCOPE

This document describes the architecture and detailed design of the *DFX* application software including its functionality, components, data structures, high-level interfaces, method of execution, and underlying hardware.

## AUDIENCE

This document is intended for all engineering and management staff concerned with the design and requirements of all IDC software in general and of *DFX* in particular. The detailed descriptions are intended for programmers who will be developing, testing, or maintaining *DFX* applications.

## RELATED INFORMATION

The following documents complement this document:

- *Database Schema* [IDC5.1.1Rev2]
- *Configuration of PIDC Databases* [IDC5.1.3Rev0.1]
- *IDC Processing of Seismic, Hydroacoustic, and Infrasonic Data* [IDC5.2.1]
- *Configuration of PIDC Processing Data Files* [IDC6.2.4]

See “References” on page 453 for a list of documents that supplement this document. The following UNIX manual (man) pages apply to the existing *DFX* software:

- *DFX* (1)
- *libscheme* (3)
- *libhydro* (3)
- *libinfra* (3)

## USING THIS DOCUMENT

This document is part of the overall documentation architecture for the IDC. It is part of the Software category, which describes the design of the software. This document is organized as follows:

- Chapter 1: Overview  
This chapter provides a high-level description of *DFX* applications, including its functionality, components, background, status of development, and current operating environment.
- Chapter 2: Architectural Design  
This chapter describes the architectural design of *DFX*, including its conceptual design, design decisions, functions, and interface design.
- Chapter 3: Shared Functionality  
This chapter describes the implementation of functions common to the various *DFX* applications including database operations, initializing site information, reading waveforms, and other common Scheme tasks.

## ▼ About this Document

## ■ Chapter 4: Seismic Processing

This chapter describes the detailed design of the eight *DFX* seismic processing applications: Automatic Seismic Detection, Automatic Origin Beam, Interactive Beam-on-the-Fly, Interactive Seismic Recall, Automatic Seismic Recall, Automatic Depth-Phase SNR, Automatic Noise Amplitude Estimation, and Automatic Seismic Event Characterization.

## ■ Chapter 5: Hydroacoustic Processing

This chapter describes the detailed design of the three *DFX* hydroacoustic processing applications: Automatic Hydroacoustic Detection, Interactive Hydroacoustic Recall, and Automatic Hydroacoustic Event Characterization.

## ■ Chapter 6: Infrasonic Processing

This chapter describes the detailed design of the single *DFX* infrasonic processing application: Automatic Infrasonic Detection.

## ■ Chapter 7: Administrative Processing

This chapter describes the detailed design of the two *DFX* administrative processing applications: Segment Archiving and Quality Control Statistics.

## ■ Chapter 8: Database Description

This chapter describes the database tables, attributes, and corresponding schema that each *DFX* application uses.

## ■ References

This section lists the sources cited in this document.

## ■ Glossary

This section defines the terms, abbreviations, and acronyms used in this document.

## ■ Index

This section lists topics and features provided in the document along with page numbers for reference.



## Conventions

This document uses a variety of conventions, which are described in the following tables.

Table I shows the conventions for data flow figures. Each application's design chapter contains two or more figures that show the architecture of the application or function. The figures primarily show control flow and external data flow. Internal data is represented only as input to functions.

Process symbols are numbered with the exception of trivial or self-evident processes. The top-level processes are numbered as [X.Y], where X is the application number (see Table 1 on page 10) and Y is the process number. These numbers have corresponding sections in the general descriptions of the processes.

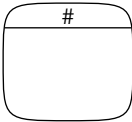
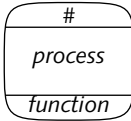
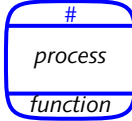




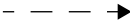
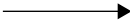
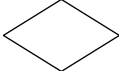
Additionally, each process symbol has one or two textual elements: The center text describes the process, and the bottom text specifies the name of the function used. If the process consists of several functions or inline Scheme code, the bottom text is left blank.

Process symbols that are highlighted (via thick lines) have an additional detailed description. The application and process numbers are included in the heading of the detailed description and are shown in the Table of Contents. Each detailed description has an architectural diagram showing the subprocesses that make up the function. The subprocesses are numbered as [X.Y.Z], where X is the application number, Y is the process number, and Z is the subprocess number. Each of these subprocesses is described in a corresponding subsection.

Table II lists typographical conventions.

▼
About this Document

TABLE I: DATA FLOW SYMBOLS

Description	Symbol <sup>1</sup>
DFX application (shaded in Chapters 1–3)	
process (left) process that is decomposed in a later figure (right)	 or 
external source or sink of data (left) duplicated external source or sink of data (right)	 
data store (left) duplicated data store (right) M = memory store D = disk store Db = database store	 
control flow	
data flow	
decision or loop	

1. Most symbols in this table are based on Gane-Sarson conventions [Gan79].

TABLE II: TYPOGRAPHICAL CONVENTIONS

Element	Font	Example
database table	<b>bold</b>	<b>arrival</b>
database table and attribute, when written in the dot notation		<b>affiliation.net</b>
database attributes	<i>italics</i>	<i>lddate</i>
processes, software units, and libraries		<i>DFX</i>
user-defined argument names and variables used in parameter (par) files or pro- gram command lines		<i>delete-remarks object</i>
titles of documents		<i>Database Schema</i>
function names in process symbols		<i>get-infra-waveforms</i>
objects and containers		<i>wfdata</i>
computer code and output	<b>courier</b>	(qc-waveforms wfdata-con qc-rec)
filenames, directories, and websites		DFX-evch.scn
text that should be typed in exactly as shown		edit-filter-dialog
strings replaced with appropriate value	<i>&lt;value&gt;</i>	DFX <i>&lt;par_name&gt;</i>



## Chapter 1: Overview

This chapter provides a general overview of the *DFX* software and includes the following topics:

- Introduction
- Functionality
- Identification
- Status of Development
- Background and History
- Operating Environment

# Chapter 1: Overview

## INTRODUCTION

The software of the IDC acquires time-series and radionuclide data from stations of the International Monitoring System (IMS) and other locations. These data are passed through a number of automatic and interactive analysis stages, which culminate in the estimation of location and in the origin time of events (earthquakes, volcanic eruptions, and so on) in the earth, including its oceans and atmosphere. The results of the analysis are distributed to States Parties and other users by various means. Approximately one million lines of developmental software are spread across six computer software configuration items (CSCIs) of the software architecture. One additional CSCI is devoted to run-time data of the software. Figure 1 shows the logical organization of the IDC software. The Automatic Processing CSCI processes data through the following computer software components (CSCs):

- Station Processing  
This software scans data from individual time-series stations for characteristic changes in the waveforms (detection of onsets) and characterizes such onsets (feature extraction). The software then classifies the detections as arrivals in terms of phase type.
- Network Processing  
This software combines arrivals from several stations originating from one event and infers the location and time of its origin.
- Post-location Processing  
This software computes various magnitude estimates and selects data to be retrieved from auxiliary stations.

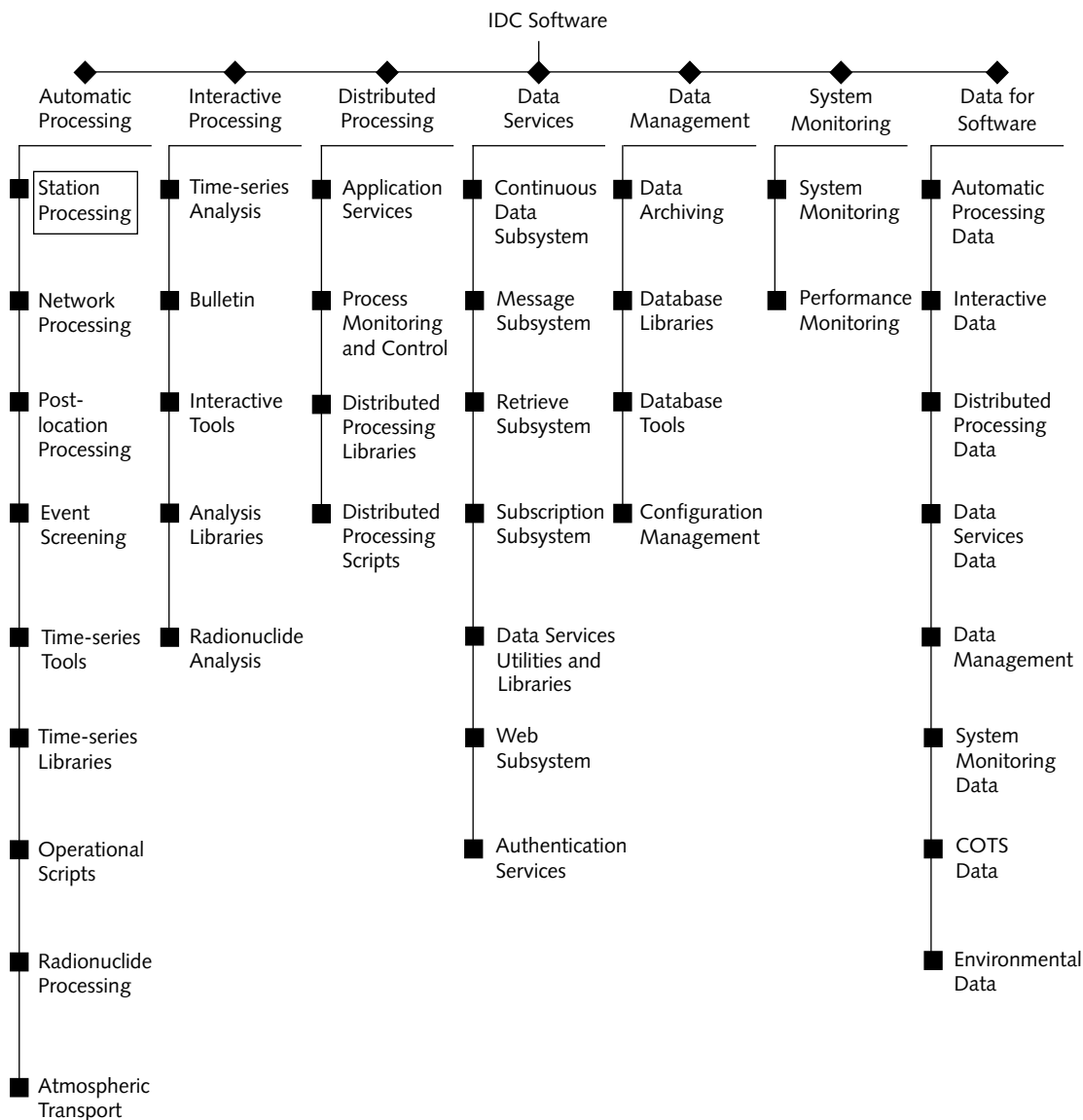


FIGURE 1. IDC SOFTWARE CONFIGURATION HIERARCHY

## ▼ Overview

## ■ Event Screening

This software extracts a number of parameters that characterize an event; then a default subset of the calculated Event Characterization Parameters eliminates the events that are clearly not explosions.

## ■ Time-series Tools

This software includes various utilities for the Seismic, Hydroacoustic, and Infrasonic (S/H/I) processing system.

## ■ Time-series Libraries

This software includes shared libraries to which several modules of the S/H/I processing system are linked.

## ■ Operational Scripts

This software provides miscellaneous functionality to enable automatic processing to function as a system.

## ■ Radionuclide Processing

This software includes the automated analysis, categorization, and flagging processes for radionuclide data.

## ■ Atmospheric Transport

This software includes the forward and backward modeling of the transport of particulates by atmospheric movements.

*DFX* is part of the Station Processing CSC of the Automatic Processing CSCI. However, there are 14 *DFX* applications for IDC processing and these are used in several places; therefore, Figure 1 does not reflect *DFX*'s entire relationship with the rest of the IDC software. Most of the applications reside in the station processing, network processing, post-location processing, and event screening categories of the Automatic Processing CSCI. There are also *DFX* applications that are used by the interactive tools in interactive processing. Figures 2, 3, and 4 show the relationships of *DFX* to the other software components.



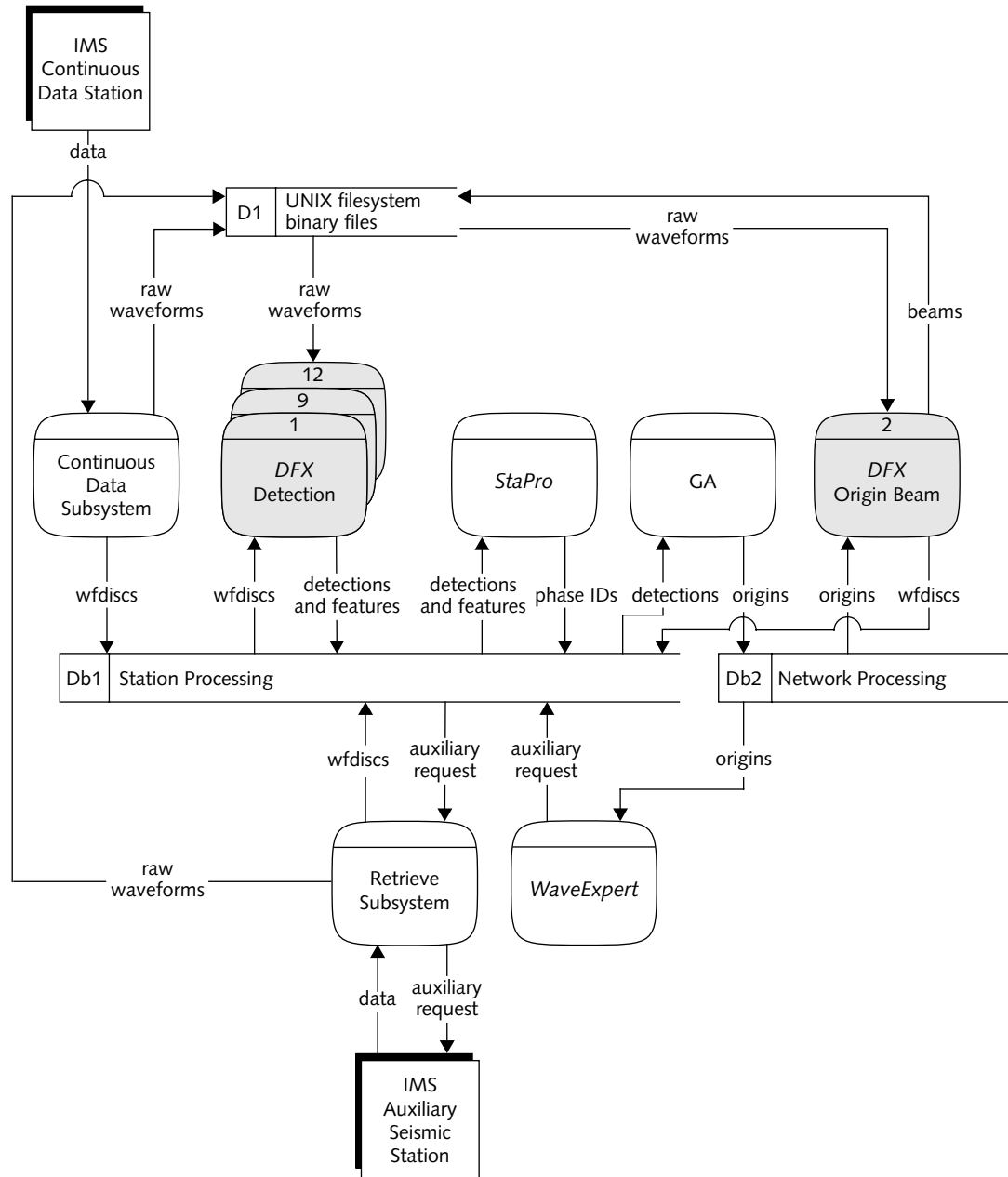
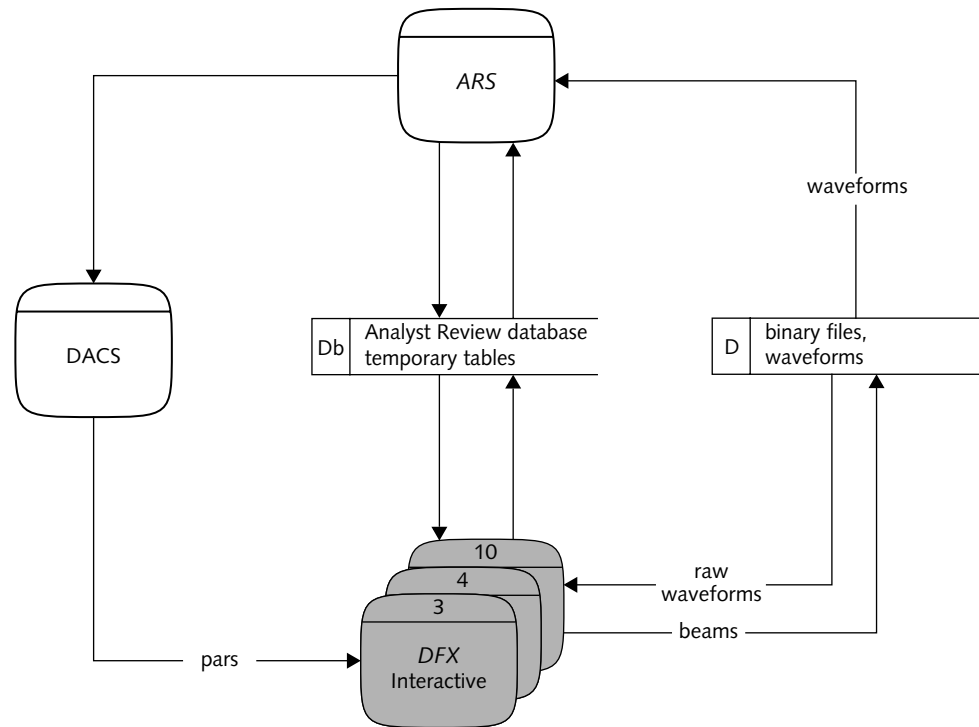


FIGURE 2. RELATIONSHIP OF DFX TO OTHER SOFTWARE UNITS IN AUTOMATIC PROCESSING

## ▼ Overview

The automatic processing pipeline (Figure 2) includes four *DFX* applications (shaded): three automatic detection and feature extraction programs, one each for seismic, hydroacoustic, and infrasonic detection processing, and the *DFX* origin beam application. The numbers in the process symbols in Figure 2 correspond to the application numbers listed in Table 1 on page 10. Automatic processing begins with continuous waveforms arriving from the IMS primary stations through the Continuous Data Subsystem. The detections and features are measured in the *DFX* detection applications, which include: *Automatic Seismic Detection* [1], *Automatic Hydroacoustic Detection* [9], and *Automatic Infrasonic Detection* [12]. In addition to feature extraction, the seismic and infrasonic *DFX* applications produce and save detection beams. The next process in the pipeline, *StaPro*, classifies detections into phase types based on the extracted waveform features. *StaPro* can also make single-station locations at seismic stations. The detections are combined in network processing by the Global Association (GA) applications to hypothesize events and estimate locations. The *DFX* origin beam application, *Automatic Origin Beam* [2], beamforms seismic waveforms based on the event locations. The beams are saved for use in interactive analysis. The event hypotheses are also used by the *WaveExpert* application to form data requests for auxiliary seismic stations. The Retrieve Subsystem acquires the auxiliary seismic data based on the requests formed by *WaveExpert*. These data are then used to refine event locations.

The *Analyst Review Station* (ARS) in interactive processing (Figure 3) can invoke three *DFX* applications (shaded): *Interactive Beam on the Fly* [3], *Interactive Seismic Recall* [4], and *Interactive Hydroacoustic Recall* [10]. ARS contains functions that create a message that is sent to the Distributed Application Control System (DACS). The message contains information specifying the data to be processed, the *DFX* application to invoke, the database account, and temporary database tables. The DACS initiates *DFX* using the message received from ARS. The primary data interfaces between ARS and *DFX* are the temporary database tables. Waveforms are read directly from the UNIX filesystem. Any results *DFX* produces are written to the temporary database tables (beams are written to the filesystem). After *DFX* completes execution, control is returned to ARS by the DACS. Finally, ARS retrieves the *DFX* results from the temporary database tables.



**FIGURE 3. RELATIONSHIP OF DFX TO OTHER SOFTWARE UNITS IN INTERACTIVE PROCESSING**

DFX is used throughout automatic post-analysis processing (Figure 4), which is initiated upon the completion of the analyst review as indicated by the application *Analyst\_Log*. The process control shown in Figure 4 is actually accomplished by the DACS, which has been left out of the figure for simplicity. Automatic post-analysis processing consists of three pipelines. The first pipeline, automatic recall processing, contains five processes, three of which are DFX applications (shaded): *Automatic Seismic Recall* [5], *Automatic Depth-phase SNR* [6], and *Automatic Noise Amplitude Estimation* [7]. After this pipeline finishes, a final human review is conducted. If the data are found acceptable, control is passed on to two pipelines: Event Characterization and Segment Archiving. The results are used to create the Standard Event Bulletin (SEB), a standard product of the IDC. The event characterization measures are made by two DFX applications (shaded): *Automatic Seismic*

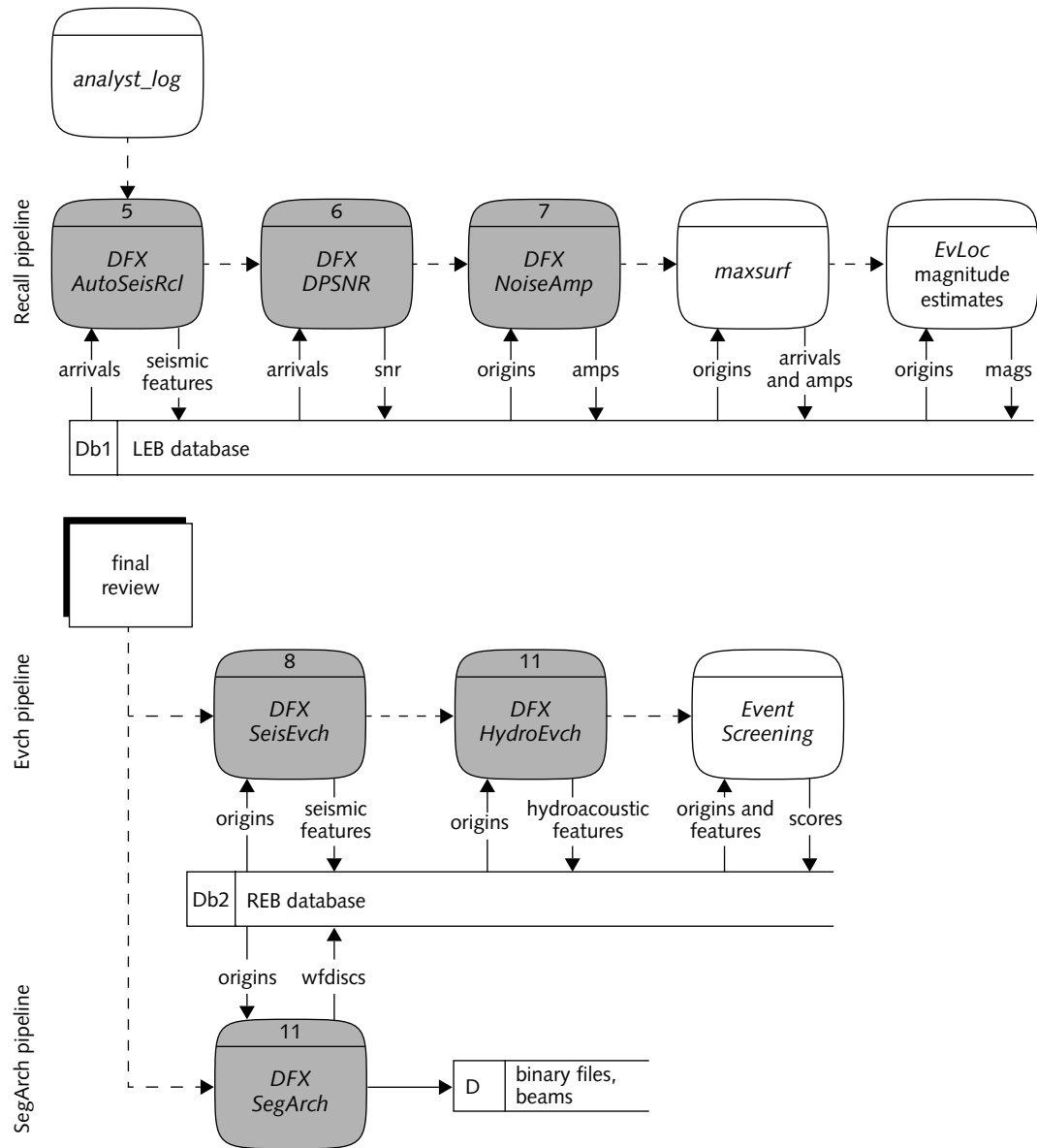
*Event Characterization* [8] and *Automatic Hydroacoustic Event Characterization* [11]. Some of the results of these processes are used by the *Event Screening* process to create scores that can be used to screen out events that are of clear natural origin and are used to create the Standard Screened Event Bulletin (SSEB). The segment archiving pipeline consists of a single *DFX* application: *Segment Archiving* [13]. This process creates an archive of the data used to form the event hypotheses in the Reviewed Event Bulletin (REB) by creating origin beam segments at predicted arrival times.

## FUNCTIONALITY

*DFX* applications perform a variety of tasks. Their primary functions are to make detections and to measure features from waveforms. *DFX* processes data from all three waveform-based technologies (seismic, hydroacoustic, and infrasonic). In the current system, *DFX* is used in automatic station processing, interactive analysis, and automatic post-analysis processing. In automatic station processing, *DFX* detects transient signals and estimates features in the vicinity of these detections. In interactive analysis, *DFX* estimates or updates features for detections that have been modified or added by the analyst. In automatic post-analysis processing, *DFX* makes a final update of the detection features, and it makes measurements, based on event hypotheses, which can be used to characterize the event. *DFX* is also used to beamform array data. The beams are saved to the UNIX filesystem for use by other applications. *DFX* also performs some administrative tasks such as archiving data associated to the final event bulletins and computing statistics on waveform quality.

## IDENTIFICATION

The 14 *DFX* applications are identified in Table 1. The number is used throughout this document to identify the application. Table 2 matches the *DFX* applications to their common abbreviations and the actual file names containing the Scheme code.



**FIGURE 4. RELATIONSHIP OF DFX TO OTHER SOFTWARE UNITS IN AUTOMATIC POST-ANALYSIS PROCESSING**

TABLE 1: DFX APPLICATIONS

Number/Application Name	Description
1 <i>Automatic Seismic Detection</i>	makes detections and measures features on seismic data and writes detection beams
2 <i>Automatic Origin Beam</i>	creates beams using theoretical delays determined from origins
3 <i>Interactive Beam on the Fly</i>	allows an ARS user to create new beams
4 <i>Interactive Seismic Recall</i>	allows an ARS user to measure features for added seismic arrivals
5 <i>Automatic Seismic Recall</i>	measures features for analyst-added seismic arrivals
6 <i>Automatic Depth-phase SNR</i>	estimates snr of seismic depth phases
7 <i>Automatic Noise Amplitude Estimation</i>	estimates seismic noise level at theoretical arrival times
8 <i>Automatic Seismic Event Characterization</i>	uses seismic data to measure various properties of events
9 <i>Automatic Hydroacoustic Detection</i>	makes detections and measures features on hydroacoustic data
10 <i>Interactive Hydroacoustic Recall</i>	allows an ARS user to update features for hydroacoustic arrivals
11 <i>Automatic Hydroacoustic Event Characterization</i>	uses hydroacoustic data to measure various properties of events
12 <i>Automatic Infrasonic Detection</i>	makes detections and measures features on infrasonic data and writes time-series that are based on signal coherence
13 <i>Segment Archiving</i>	computes beams and archives data associated to events
14 <i>Quality Control Statistics</i>	makes statistical measures of waveform quality estimates

**TABLE 2: ABBREVIATIONS AND SYSTEM NAMES FOR DFX APPLICATIONS**

Number/Application Name	Abbreviation	File Name
1 <i>Automatic Seismic Detection</i>	<i>SeisDet</i>	DFX-detection.scm
2 <i>Automatic Origin Beam</i>	<i>OriginBeam</i>	DFX-originbeam.scm
3 <i>Interactive Beam on the Fly</i>	<i>BOTF</i>	DFX-botf.scm
4 <i>Interactive Seismic Recall</i>	<i>IntSeisRcl</i>	DFX-int-recall.scm
5 <i>Automatic Seismic Recall</i>	<i>AutoSeisRcl</i>	DFX-recall.scm
6 <i>Automatic Depth-phase SNR</i>	<i>DPSNR</i>	DFX-depth-phase-snr.scm
7 <i>Automatic Noise Amplitude Estimation</i>	<i>NoiseAmp</i>	DFX-noiseamp.scm
8 <i>Automatic Seismic Event Characterization</i>	<i>SeisEvch</i>	DFX-evch.scm
9 <i>Automatic Hydroacoustic Detection</i>	<i>HydroDet</i>	DFX-hydro-detection.scm
10 <i>Interactive Hydroacoustic Recall</i>	<i>IntHydroRcl</i>	DFX-hydro-recall.scm
11 <i>Automatic Hydroacoustic Event Characterization</i>	<i>HydroEvch</i>	DFX-evch-hydro.scm
12 <i>Automatic Infrasonic Detection</i>	<i>InfraDet</i>	DFX-infra-detection.scm
13 <i>Segment Archiving</i>	<i>SegArch</i>	DFX-segarch.scm
14 <i>Quality Control Statistics</i>	<i>QCStats</i>	DFX-qcstats.scm

## STATUS OF DEVELOPMENT

This document describes mature software that has been used operationally since 1996 at the Prototype International Data Centre (PIDC). All but three of the *DFX* applications listed in Table 1 were included in Release 1; *Automatic Depth-phase SNR* (6), *Automatic Hydroacoustic Event Characterization* (11), and *Quality Control Statistics* (14) were introduced in Release 3. The software is undergoing limited enhancements on a continuing basis.

## BACKGROUND AND HISTORY

D. Wahl of Science Applications International Corporation, Monitoring Systems Operation, developed *DFX* to replace the *SigPro* and *Beamer* Fortran applications. These were components in the early PIDC system and provided signal detection and beam generation for display purposes.

The *DFX* design was inspired by the example of other system applications such as *ARS* and *Map* that use C-code “building blocks” to provide basic processing functionality to the Scheme command-language environment. Higher level applications and functionality are constructed with the Scheme language. This architecture makes it easy to develop and customize applications without requiring recompilation or changes to the program code. With this design philosophy *DFX*, a single program, can provide a broad spectrum of signal-processing tasks and is expandable as new system requirements arise.

*DFX* was first used operationally in 1996. It is a current software element of the PIDC at the CMR in Arlington, Virginia, U.S.A., and the International Data Centre of the Comprehensive Nuclear-Test-Ban Treaty Organization (CTBTO) in Vienna, Austria. Similar versions of *DFX* are used by other monitoring systems:

- United States National Data Center
- Spanish Earthquake Monitoring System
- South Korean Earthquake Monitoring System



## OPERATING ENVIRONMENT

The following paragraphs describe the hardware and commercial-off-the-shelf (COTS) software required to operate *DFX*.

### Hardware

*DFX* is designed to run on UNIX workstations. *DFX* applications are fairly resource-intensive, and typically multiple instantiations of *DFX* (in the case of station detection processing, for example) must be distributed over several processors to process all of the data from the IMS network in a timely manner. The hardware required depends on many factors such as the amount and type of data and the number of detections per day. A small seismic network can be processed on a single machine such as a Sun Ultra-5, but a large system, such as the IDC, needs high-speed, multiple-processor-type systems like a Sun Enterprise. The required disk space will scale with both the number of beams computed and the duration that its log-file messages are archived. *DFX* must have access to the database and UNIX filesystem.

### Commercial-Off-The-Shelf Software

The software is designed to run using Solaris 7 and ORACLE 8.1.5.



## Chapter 2: Architectural Design

This chapter describes the architectural design of *DFX* and includes the following topics:

- Conceptual Design
- Design Decisions
- Functional Description
- Interface Design

## Chapter 2: Architectural Design

### CONCEPTUAL DESIGN

The term *DFX* is often used to refer to the combination of two distinct entities: the *DFX* programming environment and a *DFX* application.

The *DFX* programming environment is a general purpose signal-processing tool. It consists of an application programming interface (API), special internal data representations, and a large set of functions for data manipulation and processing. It also provides functions to interface with a relational database.

A *DFX* application is a script written in Scheme (a dialect of the LISP programming language) that instructs the *DFX* API on how to process the data. The *DFX* programming environment can be thought of as a Scheme interpreter with built-in signal processing and database functionality. The phrase "run *DFX* on these data" actually means "invoke the *DFX* programming environment for the specified data using a Scheme script (not specified in the original statement)." This document primarily describes the design of the *DFX* applications (Scheme scripts) used in operations to perform a variety of functions.

The *DFX* API adds several important features to standard Scheme. Application and site configuration parameters are specified following *libpar* conventions. The *DFX* programming environment provides several mechanisms to store and manipulate data internally in addition to the standard variables (or lists) in Scheme. The two most important internal data representations are the C variable datastore (CVAR, pronounced see-var) and generic objects (GObj, pronounced gee-obj). The *DFX* API also provides an interface to a relational database through *libgdi*, and it has many general and specialized signal-processing functions.

Figure 5 shows the external data exchange used by the *DFX* API. The API first loads the application (Scheme) file (a). This file contains the high-level control instructing *DFX* on how to process the data. Next, the API loads the application and station parameters either from the command line or from ASCII files (par files) located in the filesystem (b). Generally, the Scheme application file instructs *DFX* to acquire various information from the database (c). The acquired data usually include pointers to the waveforms that are to be processed. The waveforms are read from binary files on the filesystem (d). If needed, *DFX* loads the travel-time tables that are also stored on the filesystem (e). After processing, *DFX* writes the results from processing to the database (c). If the application produces beams, then these are written to binary files on the filesystem (d). Finally, *DFX* writes log messages to ASCII files recording a variety of information about each process that is performed (f).

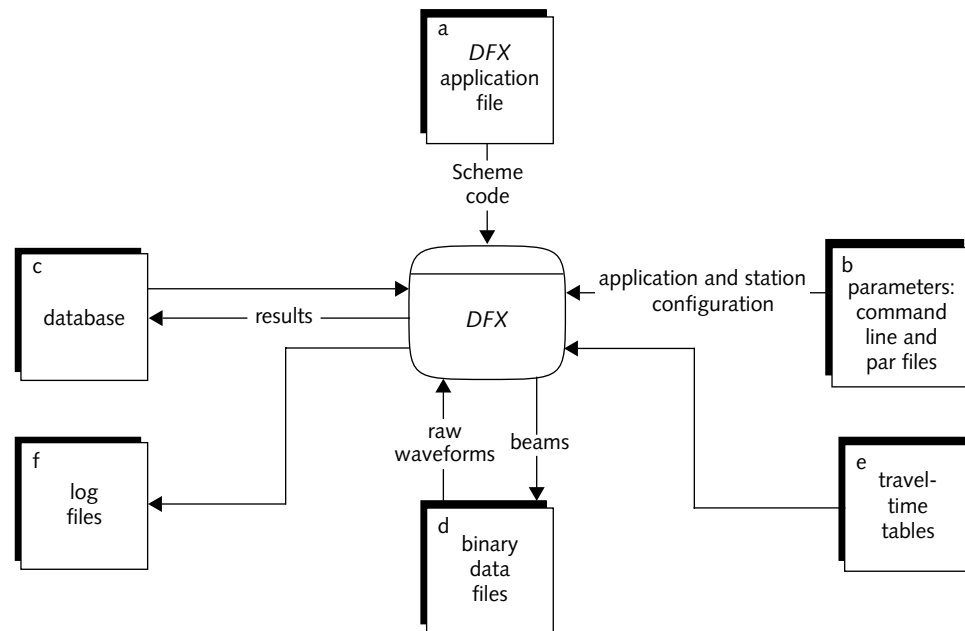


FIGURE 5. DFX EXTERNAL DATA INTERFACE

## ▼ Architectural Design

The following sections describe the data representation in *DFX*, the types of functions in *DFX*, the physical design of *DFX* Scheme applications, and invocation of a *DFX* Scheme application.

**DFX Data Representations**

Figure 6 shows *DFX*'s internal data representations and their relationship to the external data sources. *DFX* has four major types of external data: ASCII parameter files (a), relational database tables (b), binary waveforms (c), and ASCII travel-time tables (d). These data are loaded into three different types of internal memory. The parameters in the par files are loaded directly into the CVAR datastore (M2) (see "CVAR Datastore" on page 20). The database tables and the waveforms (M1) are placed in generic objects. The travel-time tables are loaded into data structures defined in *libloc* (M4). A fourth type of internal memory is Scheme variables or lists (M3). The Scheme lists are largely used for data manipulation and process control. Because the waveforms are stored in generic objects and the recipe parameters stored in the CVAR are also placed in generic objects, most of the data exchange in and out of the signal-processing functions is through generic objects. The data exchange with the relational database management system (RDBMS) is also handled with generic objects.

The sections that follow briefly describe the concept of parameters, the CVAR datastore, and generic objects.

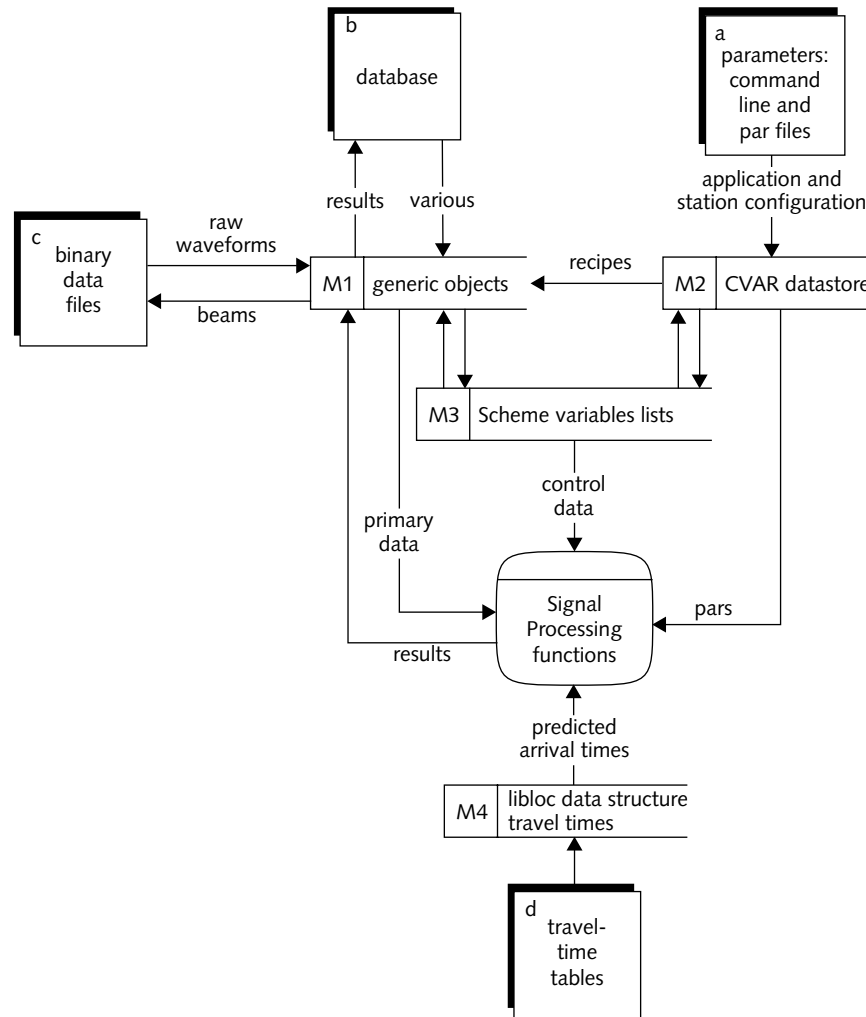


FIGURE 6. DFX EXTERNAL DATA SOURCES AND INTERNAL DATA REPRESENTATIONS

## ▼ Architectural Design

**Parameters**

DFX uses parameters (pars) and parameter files (par files) to configure applications and station-dependent attributes. The pars are used to define a myriad of values such as station name, type of filter, and detection threshold. The parameters are assigned a specific value either on the command line or in a par file. The par file is a set of parameter definitions.

The parameters' values are assigned by the notation *parameter\_name=value*. To load a par file, the command *par=file* is used either on the command line or inside another par file. Often the values or filenames used to define pars are made by concatenating other pars. This provides a mechanism to use different par files based on site-specific information that is provided at run-time.

The par files can also be used to define tables. The tables are delineated with the key symbols *#!BeginTable table-name* and *#!EndTable*. The first row in a table contains the names of the columns, which are delineated with the pipe symbol "*|*". The column order and number are not formatted. The name of the table must be as shown for the particular set of parameters described because the tabular format is converted to a common par format internally as *table-name.column-name[row-number]*. Thus, the tabular format may be entered in long-hand notation just as any other par. Normal parameter specification can be mixed with tabular parameter specification within a single file.

**CVAR Datastore**

DFX stores all pars and UNIX environment variables internally in the CVAR datastore (sometimes referred to as the CVAR table). During operation, the CVAR datastore contains the current instance of all pars that have been set. Except for a few parameters, the CVAR datastore is intended as transitory, not historical, storage. The CVAR datastore will generally contain many sets of related parameters referred to as recipes. These parameters are first transferred to generic objects before being used by the signal-processing functions.



## Generic Objects

*DFX* represents data internally using objects as defined by the common library *libgobj*. These objects are used as a common representation of all types of data that *DFX* can manipulate. Objects are data structures that encapsulate related sets of information. The information is stored in the object's attributes. An object class defines the attributes for any instance of an object in that class.

The library *libgobj* is fully integrated with *DFX* so that object classes may be defined, objects may be instantiated, and objects may be manipulated using Scheme functions. These functions may have objects as input arguments and as their return values.

Many classes of objects are used within a *DFX* application. These may be standard classes defined within *libgobj*, compiled within *DFX*, or defined within the *DFX* Scheme application.

One general category of generic objects is recipes. These objects encapsulate a set of parameters used to configure *DFX* library functions. The recipe parameters are set in the CVAR datastore from par files in the *DFX* configuration directory tree. These par files are called recipe files. The parameters are extracted from the CVAR datastore and placed into a recipe object by the *DFX* Scheme functions `init-<recipe_name>-recipe`. An example of a recipe is the quality-control recipe (*qc-rec*), which contains the parameters that tell *libqc* how to check and process the waveforms.

A common type of generic object is the container object, which is used to hold many generic objects of the same class. For example, a *wfdata* container is actually a collection of *wfdata* objects. This is similar to an array of structures that might be used in a C program. It is even possible to have a container of containers. Containers provide a convenient method to store and manipulate the data and are used throughout the *DFX* Scheme applications. More information on containers is provided in "Defining Generic Objects" on page 71.

## DFX Functions

The functions of the *DFX* Scheme applications are categorized based on the location of the function's code as follows: intrinsic functions, global functions, library functions, and local functions. These categories are described in the sections that follow.

### Intrinsic Scheme Functions

The Scheme interpreter used by *DFX* has a relatively small set of built-in, or intrinsic, functions. The most common functions are `car`, `cadr`, `define`, `let*`, `lambda`, `set!`, `null?`, `list`, `begin`, `if`, `or`, `and`, `not`, `eqv?`, `string-append`, and `map`. For information about the Scheme language that describes these functions, see one of the many books on this subject, including [Abe96] and [Fri97].

### Global Scheme Functions

*DFX* has many built-in Scheme functions that perform common tasks, including database operations, managing generic objects, and signal processing tasks. These functions are defined in the `DFXdefault.scm` file, which is loaded into every invocation of *DFX*. They are referred to throughout this document as global Scheme functions.

### DFX Library Functions

The main signal-processing functions are programmed in C and reside in *DFX* libraries. This is done to make the computationally expensive processes as efficient as possible. The C libraries contain wrapper functions that provide an interface to Scheme. The libraries define functions that *DFX* binds to Scheme symbols. To the application, these symbols appear to act the same as other Scheme functions. There are 19 local *DFX* libraries: *libamp*, *libbeam*, *libcomplexity*, *libdata*, *libdb*, *libdetect*, *libfk*, *libfs*, *libhydro*, *libinfra*, *libio*, *libonset*, *libpmcc*, *libqc*, *libsplp*, *libtf*, *libthreec*, *libuser*, and *libutil*.

## Local Scheme Functions

*DFX* applications define many functions within their Scheme files, which remain local to the application. These functions are generally specialty functions used only by the application. Sometimes local functions are defined to make the process flow easier to follow. In addition, several types of functions are used by many applications, but are specialized enough that a global version could not be efficiently created. These functions are made local to the application. For example, there are five distinct versions of the function `init-net-sites`.

## Structural Design of Scheme Scripts

The following sections describe the general tasks shared by most *DFX* Scheme applications and the general structure of the Scheme scripts.

### General Algorithm

*DFX* uses Scheme to control processing and data flow. A *DFX* Scheme application initiates many operations, including these common ones:

- read application parameter data
- read site-specific parameter files
- read information from the database
- read waveforms from the filesystem
- process data via *DFX* C library functions
- save results to the database

Process control is generally more complicated than indicated by these operations due to loops over various data objects and complicated signal-processing functions. However, almost all applications perform these operations at some time. (See the individual applications' design sections in this document for the exact operations used by each application.)

## ▼ Architectural Design

**General Structure of Scheme Files**

Because Scheme is an interpretive language, functions must be loaded into memory prior to being executed. Therefore, the general structure of a Scheme application is as follows:

```
;; beginning of Scheme file
define function1 (...)
define function2 (...)
...
define functionX (
    ...
    (function1)
    ...
    (function2)
    ...
)
(functionX)
;; end of Scheme file
```

In a compiled programming language such as C `functionX` would be considered the main program. The last statement in the file, `(functionX)`, tells *DFX* to execute the program.

**Scheme Application Invocation****Configuration**

In general, a *DFX* application consists of a Scheme script and two application par files. The Scheme file has the name `DFX-<application>.scm` and the par files have the names `DFX-<application>.par` and `DFX-site-<application>.par`. Some applications will share either or both of the par files with other applications.

The application's par file (DFX-*<application>.par*) specifies general parameters that define the system's configuration. It usually references standard DFX par files containing system-specific parameters. The parameters specifying the database and database account and a variety of other general parameters are also set in the application's par file. The name of the site par file (DFX-site-*<application>.par*) and the name of the Scheme script file (DFX-*<application>.scm*) are defined in the application's par file.

The site par file uses the par command (par=*<recipe\_file>*) to read site-specific recipe files. The filenames (*<recipe\_file>*) are often character-strings created by concatenating site-specific parameters (such as *sta*, *NetType*, *StaType*, and *WaveType*), and thus different recipe files can be read for different stations. Most applications use a large number of recipe files that cover a variety of functions. (See [IDC6.2.4] for more information.)

### Executing Scheme Applications

A DFX Scheme application is initiated by the UNIX command line:

```
DFX <par_name>=<par_value> ... par=DFX-<application>.par
```

The user must set the parameters specified on the command line (above, *<par\_name>*) to the particular data being processed. For example, the detection applications use the station code, the start-time, and the end-time to define the waveforms to process. The input parameter tables (the first table in the description of each application in this document) list the parameters that are expected to be specified on the command line.

## DESIGN DECISIONS

The following design decisions pertain to DFX.

## ▼ Architectural Design

**Programming Language**

The goal of being able to flexibly configure standard signal-processing components to satisfy several distinct operational requirements lead to the decision to write applications in an interpretive language so that major signal processing units could be rearranged without having to recompile the entire system. However, due to the computationally intensive nature of many of *DFX*'s signal processing functions, some functions had to be written in a compiled programming language.

Scheme was chosen as the interpretive language for *DFX* because of its flexibility and its convenient memory allocation [Abe96] and [Fri97]. It is also used to configure other IDC applications such as *ARS* and *Map*. The signal-processing functions are written in the C programming language, which was chosen for its efficiency, maintainability, and portability.

Scheme's memory management has many advantages over languages such as C or FORTRAN. For example, the Scheme programmer does not need to explicitly allocate and de-allocate variables. The Scheme environment handles all memory functions; consequently an entire range of common problems related to memory management is eliminated. For example, many versions of FORTRAN allow the programmer to "write" outside the bounds of an allocated array during run-time, corrupting data or program instructions. The situation can be worse in the C language where the programmer is responsible for directly allocating and freeing memory. Here the programmer can free unallocated memory, allow memory to pass out of scope—known as leaking memory—and introduce a host of other memory-related problems, which are often difficult to diagnose.

Because the Scheme environment manages the memory, it provides a low-risk approach for scientific programmers to design or modify the sequence and connectivity of the *DFX* data handling and signal processing library functions.

**Global Libraries**

The software of *DFX* is linked to the following shared libraries: *libscheme*, *libgobj*, *libdataqc*, *libbeam*, *libfk*, *libfilter*, *libgdi*, *libtable*, *libpar*, *libprob*, *libspectra*, *libwav*, *libwfm*, *libloc*, *libLP*, *libgeog*, *libinterp*, *libresponse*, *libaesir*, and *libstdtime*.

## Database

The RDBMS acts as the data storage center for *DFX* results. *DFX* uses the functions in *libgdi* to interact with the database. The database used by *DFX* is the same as for all of the other software units at the IDC. Different *DFX* applications read and write different tables as identified in Table 3 and Table 229.

## Interprocess Communication (IPC)

*DFX* does not have true IPC capability. Control flow is achieved through the *tuxshell* application. However, data transfer between applications can be accomplished indirectly by database tables. Some data (such as the names of database tables) can also be passed to *DFX* by the DACS through command-line parameters. Binary waveforms are read directly from the filesystem using pointers obtained from the database. For example, *ARS* creates and populates temporary tables and passes the database account and table names through the DACS to the interactive *DFX* applications by command line parameters. The *DFX* application reads and writes to the temporary tables. When *DFX* finishes, control is passed back to *ARS*, and *ARS* reads the temporary tables to obtain the *DFX* results.

## Filesystem

*DFX* uses several types of files from the filesystem. The parameters used to control data processing are stored in *par* files (see [IDC6.2.4]). The Scheme commands that control processing flow are stored in the *DFX-<application>.scm* file. The actual waveforms are read from binary files located on the filesystem, and the beam data are written to the filesystem as binary files. In addition, travel-time tables are read from ASCII files in the *earth-specs/* directory of the configuration tree. *DFX* also writes informational and error messages to *stdout* and *stderr*, which are typically recorded in log files.

## ▼ Architectural Design

**Design Model**

The design of *DFX* is primarily influenced by a need for flexibility and timeliness. Signal processing is computationally intensive, and the processing time must be kept to a minimum. The combination of the efficient, low-level, C functions and the flexible Scheme developmental environment is an effective model to accomplish the signal processing requirements of the IDC system.

**Database Schema Overview**

*DFX* uses the ORACLE database for the following purposes:

- retrieve pointers to binary files that contain waveforms
- retrieve station geometry, location, and calibration information
- retrieve previous results for updating
- retrieve arrival and event related information
- save results of data processing

Table 3 shows the tables used by current IDC *DFX* applications. The Name field identifies the database table. The Mode field is "R" if *DFX* reads from the table and "W" if the system writes to the table. Because the *DFX* applications cover such a variety of functions, no single application uses all of the database tables listed here. See "Chapter 8: Database Description" on page 441 for application-specific information.

**TABLE 3: DATABASE TABLES USED BY DFX**

Name	Mode	Description
affiliation	R	groups stations into networks or arrays
amplitude	R <sup>1</sup> /W	contains arrival-based and origin-based amplitude measurements
amp3c	R <sup>1</sup> /W	contains amplitude measurements made on three-component (3-C) data for a specific detection
apma	R <sup>1</sup> /W	contains results of particle motion analysis for a specific detection



TABLE 3: DATABASE TABLES USED BY DFX (CONTINUED)

Name	Mode	Description
<b>arrival</b>	R/W	contains summary information about arrivals
<b>assoc</b>	R	contains information that connects arrivals (entries in the <b>arrival</b> table) to a particular origin
<b>ceppks</b>	R <sup>1</sup> /W	contains results of cepstral analysis and includes the amplitude and quefreny of cepstral peaks that are consistent among multiple phases associated with the same event
<b>complexity</b>	R <sup>1</sup> /W	contains the complexity event characterization parameter estimated by <i>DFX</i>
<b>detection</b>	R <sup>1</sup> /W	contains summary information about S/H/I detections
<b>hydro_features</b>	R/W	contains feature measurements for hydroacoustic data from <i>DFX</i>
<b>infra_features</b>	W	contains feature measurements for infrasonic data from <i>DFX</i>
<b>instrument</b>	R	contains ancillary calibration information
<b>lastid</b>	R/W	contains counter variables (last value used for keys)
<b>origerr</b>	R	contains summaries of confidence bounds in origin estimations
<b>origin</b>	R	contains information describing a derived or reported origin for a particular event
<b>parrival</b>	R <sup>1</sup> /W	contains the predicted arrivals and associations for origin-based amplitude measures
<b>qcstats</b>	W	contains waveform quality statistics
<b>sensor</b>	R	contains calibration information for specific sensor channels
<b>site</b>	R	contains station location information
<b>sitechan</b>	R	contains station channel information
<b>splp</b>	R <sup>1</sup> /W	contains short-period/long-period energy ratios
<b>spvar</b>	R <sup>1</sup> /W	contains the variance of the detrended log spectrum between <i>fmin</i> and <i>fmax</i> for an arrival identified by <i>arid</i>
<b>thirdmom</b>	R <sup>1</sup> /W	contains the third moment of frequency and the percentage of signal frequency amplitudes greater than the corresponding noise frequency amplitudes for an arrival identified by <i>arid</i>

## ▼ Architectural Design

**TABLE 3: DATABASE TABLES USED BY DFX (CONTINUED)**

Name	Mode	Description
<b>timefreq</b>	R <sup>1</sup> /W	contains the time-frequency measurements for event characterization
<b>wfdisc</b>	R/W	contains a waveform header file and a description
<b>wftag</b>	W	links various identifiers (for example, <i>orid</i> , <i>arid</i> , and <i>stassid</i> ) to <i>wfid</i>

1. This table is read for screening purposes only.

## FUNCTIONAL DESCRIPTION

The *DFX* applications are used to process three types of waveforms (seismic processing, hydroacoustic processing, and infrasonic processing) and to conduct some administrative processing tasks. Each of the technology groups include various applications, which generally fall into one of two categories: detection driven or event driven. Detection-driven processing measures waveform features in the vicinity of transient signals. Event-driven processing measures features within a data window predicted from event locations and a signal propagation model.

“Chapter 3: Shared Functionality” on page 39 describes the implementation design of several tasks that are common to almost all of the *DFX* applications. The following paragraphs briefly describe these shared functionality tasks, 12 *DFX* applications categorized by technology type, and 2 administrative *DFX* applications.

### Shared Functionality

Each of the *DFX* Scheme applications is designed to perform specific types of data processing, however they may perform a number of common tasks. “Chapter 3: Shared Functionality” on page 39 describes the most common tasks and how they are accomplished. Common tasks include database operations, initializing sites, reading waveforms, and common *DFX* tasks in Scheme.

## Database Operations

*DFX* reads and writes to the database as a key method for obtaining processing input and storing output. “Database Operations” on page 40 describes how *DFX* database connections are initialized and terminated and how tables are queried, updated, and written.

## Initializing Sites

Most *DFX* applications need parametric data that are specific to each site. Applications perform a process known as *Site Initialization* that retrieves the station-dependent information. This process creates a special object, called *initsite*, that contains the station-related information. The exact elements that are stored vary from application to application. Four general categories of information are stored in an *initsite* object: individual parameters, database objects, recipes, and waveform objects. “Initializing Sites” on page 46 describes how *Site Initialization* is implemented and exactly what data each application stores.

## Reading Waveforms

Most *DFX* applications operate on waveforms. A standard method of reading and managing data in *DFX* has been developed. This method includes creating special objects that hold the waveform and calibration data, reading the data from the filesystem, and checking the quality of the data. “Reading Waveforms” on page 58 describes the general methodology followed by *DFX* applications.

## Common DFX Scheme Tasks

Scheme is the standard language for *DFX* applications and, like most computer languages, there is a high amount of flexibility in how to implement various tasks. For *DFX*, standard methods have been designed for some of the more common tasks. “Common DFX Scheme Tasks” on page 71 describes many of the specific constructs used to manipulate process control and various data types.

## Seismic Processing

The majority of the *DFX* applications process seismic data. These applications include: *Automatic Seismic Detection* [1], *Automatic Origin Beam* [2], *Interactive Beam-on-the-Fly* [3], *Interactive Seismic Recall* [4], *Automatic Seismic Recall* [5], *Automatic Depth-phase SNR* [6], *Automatic Noise Amplitude Estimation* [7], and *Automatic Seismic Event Characterization* [8].

### Automatic Seismic Detection [1]

This application detects transient seismic signals and measures features in the vicinity of the detections. It also may write detection beams. The exact features estimated depend on the type of instrument and network. The application is called for a specified seismic station and processing time interval. For each detection, *DFX* inserts a record into the **detection** and **arrival** tables. Other tables that can be populated are **amplitude**, **amp3c**, **apma**, **wfdisc**, and **wftag**.

### Automatic Origin Beam [2]

This application creates beams from waveforms at a seismic station based on event locations. The application is called for a given time period and a given network of stations. The application computes time delays used to steer the beam to the event location for the appropriate phase. The beams are written to the filesystem, and records that point to the newly created beams are entered into the output **wfdisc** table.

### Interactive Beam-on-the-Fly [3]

This application is designed to allow analysts to create (or update) beams after adding or modifying an event in *ARS*. The application is called for a single origin and a list of stations. It predicts the time delays between array elements from the origin, station location, and theoretical travel-time tables for the desired phase. It applies the delays to the waveforms of the array elements and sums the time series across instruments to form the beam. These beams are written to the filesystem, and records are entered into a **wfdisc** table for *ARS* to read.

### Interactive Seismic Recall [4]

This application is designed to allow the analyst to compute features for newly created or modified arrivals. The application is called for a set of arrivals that are stored in a temporary database table and read by *DFX*. The features are similar to those estimated in automatic detection processing. The actual measures made are station dependent but can include arrival time uncertainty, best-beam calculations, f-k and polarization analysis, signal-to-noise ratios, and various amplitude measures. The tables that are updated are likewise station dependent but can include: **arrival**, **detection**, **amplitude**, **amp3c**, and **apma**.

### Automatic Seismic Recall [5]

This application is designed to automatically compute features for newly-created or modified arrivals. A variety of methods may be used to define the set of arrivals to process; however, in normal operations the set of analyst-added arrivals is used. The features measured are similar to those estimated in automatic detection processing and interactive recall processing. The actual measures made are station dependent but can include arrival-time uncertainty, best-beam calculations, results from f-k and polarization analysis, signal-to-noise ratios, and various amplitude measures. The tables that are updated are also station dependent but can include: **arrival**, **detection**, **amplitude**, **amp3c**, and **apma**.

### Automatic Depth-phase SNR [6]

This application is a post-analysis process that obtains all event-associated depth phases in a given time interval and computes a specialized signal-to-noise ratio (snr) value for them. The application is called for a given set of events specified by a time interval and station network. The depth-phase snr measure is a ratio of the amplitude in a short interval encompassing a depth-phase arrival to the amplitude in a short interval preceding the depth-phase arrival. Depth-phase snr should not be confused with the **amplitude** table's *snr* value, which is the ratio of the long-term average to a detection's maximum short-term average. This application inserts records for each depth phase into the **amplitude** table.

## ▼ Architectural Design

**Automatic Noise Amplitude Estimation [7]**

This application is designed to estimate noise amplitudes at stations that lack observed arrivals for an event. The application is called for a given time interval and station network. The amplitudes are measured over windows predicted to contain an arrival based on the station-to-event travel time. This measure places an upper limit on the arrival's amplitude at the given station and can be used in maximum-likelihood magnitude estimations. This application inserts records for each noise measurement into the **parrival** and **amplitude** tables.

**Automatic Seismic Event Characterization [8]**

This application is designed to produce measurements that can be used to characterize an event. The application is called for a given time period and station network. Measurements are made at each station for each event that exists within the given time interval. The estimates include a variety of amplitude measures, time-frequency measures, third-moment-of-frequency measures, source-multiplicity measures, first-motion measures, short-period to long-period signal measures, and complexity measures. This application inserts records into the **amplitude**, **ceppks**, **complexity**, **parrival**, **splp**, **spvar**, **timefreq**, and **thirdmom** tables.

**Hydroacoustic Processing**

Hydroacoustic processing includes three *DFX* applications: *Automatic Hydroacoustic Detection* [9], *Interactive Hydroacoustic Recall* [10], and *Automatic Hydroacoustic Event Characterization* [11].

**Automatic Hydroacoustic Detection [9]**

This application detects transient hydroacoustic signals and measures features in the vicinity of the detections. The application is called for a specified station (in the hydroacoustic network) and an interval of time. The detections are made using a STA/LTA type detector. Many features relevant to hydroacoustics are estimated for

each detection. These features include the signal onset, termination, duration, various amplitude measures, energy distribution estimates, and cepstral parameters. The features are measured over a suite of frequency bands. For each signal *DFX* inserts a record into the **detection** and **arrival** tables. *DFX* inserts a record into the **hydro\_features** table for each frequency band with measurable energy.

### **Interactive Hydroacoustic Recall [10]**

This application is designed so that hydroacoustic features can be measured after the analyst modifies the onset and termination times of a hydroacoustic arrival or adds a new arrival. The application is called from within *ARS* for a set of arrivals stored in a temporary table. The measurements include arrival times, energy distributions, and cepstral parameters. The estimation process is very similar to that in automatic hydroacoustic data processing except that it does not attempt to calculate onset and termination times. Instead it uses the times supplied by the analyst through *ARS*. It computes features for only one frequency band chosen by the analyst. *DFX* updates the records in the **arrival** and **hydro\_features** table for each arrival processed.

### **Automatic Hydroacoustic Event Characterization [11]**

This application estimates the maximum possible hydroacoustic signal for events that occur within the ocean basins. The noise level is also estimated at the predicted time of arrival, which provides a means of validating the signal integrity. The application is called for a given time interval and network of stations. The time interval defines which events to use, and the network defines the hydrophone stations to process. For each origin the application finds all hydrophone stations in the network that have a clear oceanic path to the event. It predicts an arrival window at each of these stations based on the origin error ellipse and hydroacoustic travel-time tables. It computes four amplitude measures within the predicted time window. These amplitude measures are designed to estimate the noise level and the

## ▼ Architectural Design

maximum signal level in the time window in high and low frequency bands. For each predicted arrival *DFX* inserts a record in the **parrival** table and four records into the **amplitude** table. The results are used in Event Screening.

### Infrasonic Processing

Infrasonic processing consists of a single *DFX* application: *Automatic Infrasonic Detection* [12].

#### Automatic Infrasonic Detection [12]

This application detects transient airborne infrasonic signals and estimates the numerical values of a variety of features in the temporal vicinity of the detections. This application is called for a specified infrasonic sensor array (in the infrasonic network) and an interval of time. The detections are made using a two-stage detection algorithm. The first stage develops candidate detections by applying a threshold detection statistic in a multiple hypothesis environment. The first-stage detection statistic is a measure of the spatial coherence of bandpass-filtered waveforms. The second stage declares detections by requiring time coincidence of both an STA/LTA type energy detector and a refined spatial-coherence detector.

Many features relevant to infrasonic signals are estimated for each declared detection. These features include the signal peak-energy time, back azimuth, magnitude slowness, and waveform amplitude, as well as a variety of time- and frequency-domain features. For each detected infrasonic signal *DFX* inserts a record into the **detection**, **arrival** and **infra\_features** tables. For each processing time interval *DFX* also inserts a group of three auxiliary “beams” into the **wfdisc** table, together with corresponding “.w” files in the UNIX filesystem.

### Administrative Processing

The administrative processing consists of two functions: Segment Archiving (*SegArch*) and Quality Control Statistics (*QCStats*). *SegArch* extracts pieces of data that relate to a given event for later retrieval. *QCStats* is the only application that is not directly called, but instead is called from within other *DFX* applications.



### Segment Archiving [13]

This application is designed to extract and store relevant waveforms for each event in the final reviewed bulletin. The application is called for a time interval and a network of stations. For each origin/station pair *DFX* creates an origin beam and writes it to the filesystem. *DFX* inserts records for each beam into the **wfdisc** and **wftag** tables.

### Quality Control Statistics [14]

This application is designed to summarize attributes made during processing to check the quality of the data. The application is called from within the automatic detection processing applications for the waveforms being evaluated. Statistical measures concerning the quality of the waveforms are computed and stored in the database. These measures can be used to evaluate the data quality of a given station. *DFX* inserts a variety of records into the **qcstats** table.

## INTERFACE DESIGN

This section describes *DFX*'s interfaces with other IDC systems, external users, and operators.

### Interface with Other IDC Systems

*DFX* uses the database as the primary interface with other IDC systems. Two secondary methods are also used for data exchange among applications: command line parameters and binary waveform files. The command line parameters are used to pass *DFX* the information needed to initiate processing and are typically provided through a *tuxshell* process. These parameters can include the name of the application par file, some database tables, time intervals, and station codes. The raw waveforms are read from the UNIX filesystem, and beams produced by *DFX* applications, if any, are written to the UNIX filesystem. The pointers to these waveforms are stored in the **wfdisc** table.

## ▼ Architectural Design

**Interface with External Users**

*DFX* produces many of the results that are later used to create IDC external products, but external users have no interaction with *DFX* itself.

**Interface with Operators**

*DFX* communicates with the system operator primarily through log files created during program execution. *DFX* also returns an exit status code that can be used by the DACS to alert the operator of potential problems.

## Chapter 3: Shared Functionality

This chapter describes several tasks that are shared by many or all of the individual *DFX* applications:

- Database Operations
- Initializing Sites
- Reading Waveforms
- Common DFX Scheme Tasks

## Chapter 3: Shared Functionality

### DATABASE OPERATIONS

The database is the main interface between *DFX* and other processes at the IDC. This section describes the general method that *DFX* uses to communicate with the database. The specifics can vary among applications (see “Database Description” on page 441 for database table usage of each application).

Almost all *DFX* applications use the following five basic operations when interacting with the database:

- open database
- query database
- check for duplicate objects
- update or insert objects
- close database

To open the database an application initializes a connection to the appropriate ORACLE database account. After a connection has been established, queries for data can be made from the tables contained in the account. Data are saved to the database by either updating existing records or inserting new records. Finally, the database connection is closed before exiting the *DFX* application. These steps are described in the following sections. The actual communication with the ORACLE database is accomplished through the common C libraries *libgobj* and *libgdi*. This document does not describe the internal design of these libraries, but instead describes the Scheme language steps that *DFX* applications generally follow to interact with the database.

## Opening and Closing the Database

The application must establish a connection to the database before any data can be read or written. This is accomplished using the global *DFX* Scheme function `open-db`. This function takes no arguments and is usually executed near the beginning of the *DFX* application. This function uses two parameters (*database-vendor* and *database-account*) from the CVAR datastore. The database account is a character-string with the format "*account/password@database*". Typically the database account is specified in the application's par file by a parameter defined in the *system-spec* configuration tree (see [IDC6.2.4], [IDC5.1.3Rev0.1]).

The database connection remains open until it is closed by the global *DFX* Scheme function `close-db`. If the database has been updated, then prior to closing the database one of two actions should be taken: either the additions to the database should be saved using `commit-db`, or they should be discarded using `rollback-db`. These functions (`close-db`, `commit-db`, and `rollback-db`) do not take arguments and assume that the database connection is open.

## Querying the Database

Records are retrieved from the database by the global *DFX* Scheme function `query-for-containers`. The function takes the object class name and a query character-string as input and returns an object container. The generic implementation is as follows:

```
(set! <type>-con (query-for-container
  (say-<type>-object-class) query))
```

where *<type>* is the name of the desired object, *say-<type>-object-class* is the function that provides the object's identification character-string, and *query* is a character-string containing an SQL command. The function returns an object container. There is an object in the container for every record returned by the database query.

## ▼ Shared Functionality

The query is a character-string that follows SQL conventions. It has the general form "SELECT ... FROM ... WHERE ..." (see [IDC5.1.2] for more information on the SQL language).

The objects returned by `query-for-container` are often referred to in the code as *dbobjects*. A *dbobject* is created in Scheme using the appropriate `say-<type>-object-class` function (see "Defining Generic Objects" on page 71). A *dbobject* type corresponds to a database table definition; that is, the components of the *dbobject* have a one-to-one correspondence with the attributes in the database table. For example, a *dbaffiliation* object has the attributes *net*, *sta*, *lddate* corresponding to the attributes in the **affiliation** table of the database. Other types of objects can also be used in the `query-for-container` function. For example, the *wfdata* object does not correspond to a specific table, but instead contains attributes from several tables. The `query-for-container` function will populate the components of the object that correspond to attributes listed in the SELECT command of the query.

The function `query-for-container` is usually called from a wrapper function within the DFX application. The wrapper function is named `query-for-<obj>-by-<field>` where *<obj>* is the type of the intended output object, and *<field>* is the attribute (or attributes) used to constrain the query. For example, a query function that returns a *dbsite* container based on a station list and a time would be called "`query-for-dbsite-by-sta-time`".

The main purpose of the wrapper function is to prepare the query. A query has three parts: attributes to return, tables to use, and constraints to apply. The attributes to return are chosen to correspond to the components of the object type. The table names are generally retrieved directly from the CVAR datastore. The values that constrain the query are generally passed into the wrapper as arguments. Although an object may have many components corresponding to attributes from a database table, only the attributes that are specified in the SELECT command will be populated as a consequence of the query.

## Screening Database Containers

For a variety of reasons *DFX* applications may process the same data more than once. In automatic detection processing, time windows overlap to ensure that signals that occur at the edge of a time segment are detected, but this can cause two *DFX* invocations that operate on adjacent time segments to find the same detection. Occasionally problems in operations may require that applications be re-run, or the user may re-run a portion of data offline. Any of these situations can cause a *DFX* application to form entities that already exist in the database. However, records describing the same phenomenon must not be duplicated in the database.

The process of discarding *DFX* objects because they are duplicates or because they do not meet some criterion is referred to as “screening.” Two types of screening occur in *DFX* applications: internal and external. Internal screening removes objects that are either redundant within the current execution or do not meet some criteria. This process is described in “Common *DFX* Scheme Tasks” on page 71. External screening checks for objects produced in the current run that correspond to records existing in the database. This screening process is generally conducted in applications that insert new data into the database.

*DFX* has many screening functions with various names. The convention for naming screening functions is to begin with the word *screen*. External screening functions contain the word *existing*. Screening functions’ names that do not contain the word *existing* indicate that they do internal screening.

The functions that perform the external screens take many forms, but follow the same general steps. The database table that contains the type of objects to be screened is queried for all records that could possibly match the objects created by the current application. This query is generally based on the time interval being processed or the station code. The returned database container is checked against the internal container. The criteria that establish a match varies depending on the object type. Detections are generally matched based on times within a certain tolerance whereas objects created in event characterization are generally matched based on origins and stations. When a match is found the object is removed from the current container. The container that only has objects that were not found in the database is returned to the calling function.

## ▼ Shared Functionality

Many of the screening functions are actually wrapper functions to the global *DFX* Scheme function `screen-container-by-existing-container`. This function is implemented by

```
(screen-container-by-existing-container con query
  compare)
```

where `con` is the container to be screened and `query` is a character-string constructed following the same method described in “Querying the Database” on page 41. The third argument, `compare`, is a symbol bound to a function that does the actual comparison between objects. The function `compare` takes two objects as input and returns `t` if it determines that the objects are equivalent and `nil` otherwise. If the objects are found to be equivalent, then `screen-container-by-existing-container` removes the object from the input container. The container is returned after all objects have been compared.

### Writing to the Database

When *DFX* writes its results to a database, it either modifies existing records or inserts new records depending on the application. Detection applications generally create new records while recall applications fill null-value fields of existing records.

All information submitted to the database must first be placed in a *dbobject* or a *dbobjects* container, that is, a *dbcontainer* (see “Querying the Database” on page 41 for the definition of a *dbobject*). For example, to enter a new record into the **arrival** table from a *detection* object, a *dbarrival* object must be created using the *DFX/libdb* function `create-dbarrival-from-detection`.

Applications that generate new database records and do not update records use the global *DFX* Scheme functions `submit-container-db` and `submit-object-db`. These functions insert the information from the *dbobjects* into the database. The functions are implemented with the following Scheme code:

```
(submit-container-db db<name>-con table)
(submit-object-db db<name> table)
```



where `db<name>` refers to the database objects and `table` is the name of the output database table. One database record is inserted for each object in the container.

Two methods are used in *DFX* to update the database. If all of the records to be updated already exist in the database then the application can use the SQL command `update`. This is accomplished by creating a character-string that contains the `UPDATE` command. The string has the general form “`UPDATE ... SET ... WHERE ...`” (see [IDC5.1.2] for more information on the SQL language). This command is executed by the following Scheme code:

```
(query-for-status (say-count-object-class)
  update-command)
```

where `update-command` is the SQL string. Applications that update existing records but sometimes may also insert new database records will have a function called `update-or-insert-<type>-by-<field>`, where `<type>` is either `object` or `container` and `<field>` is the database attribute used to find pre-existing records. An example would be

```
(update-or-insert-container-by-arid con table arid)
```

where `con` is the *dbcontainer*, `table` is the name of the database table to update, and `arid` is a list of arrival identifiers from the *dbcontainer*. The function first creates a character-string that contains the `WHERE` clause of an SQL command. The `WHERE` clause should specify only records that match the new container contents. The function returns the number of records in the database that match the `WHERE` clause by the global *DFX* Scheme function `say-db-table-count`. If this count is non-zero then the records are deleted from the table by the global *DFX* Scheme function `delete-from-table-where`. Finally the new records are inserted by the global *DFX* Scheme function `submit-container-db`.

Many of the entries submitted to the database are required to carry a unique (positive) integer-valued identification number, which can be used as a primary key in later database queries. A few examples of the types of these identification numbers are: *arids*, *ampids*, *parids*, and *wfids*. The *arids* are arrival identification numbers, one for each detection declared in *DFX* processing. Similarly, the *ampids* are

## ▼ Shared Functionality

amplitude identification numbers, and the *wfids* are waveform identification numbers. The numbers are initially obtained by means of the global Scheme function, `query-for-lastid`, which returns a numerical value. This value is the identification number of the *n*-th entity requested. For example, if five *arids* are requested the numerical value for the fifth (highest) *arid* is returned. It is implied that the remaining four *arids* are the four integers immediately preceding the returned value. The database table **lastid** contains the latest type (*keyname*) of identification number. To ensure that the identification numbers remain unique, the database is immediately committed after updating the *keyvalue*. Thus, the act of requesting and receiving an allocation of one or more ID numbers via `query-for-lastid` causes all pending transactions to be committed. A rollback immediately following `query-for-lastid` has no effect on the state of the database.

## INITIALIZING SITES

Most DFX applications perform the important task of site initialization. The purpose of this task is to create a special object called *initsite* that contains site-specific information. An *initsite* object is generally created for each station being processed. If more than one station is being processed, the application may either produce all of the *initsite* objects and store them in a container, or it may create and process one *initsite* object at a time. The exact information that is stored varies from application to application. Four general categories of information are stored in an *initsite* object: individual parameters, database objects, recipes, and waveform objects. These categories and the methods used to produce the *initsite* objects are described further in the following sections.

### Contents of Initsite Objects

The *initsite* object may contain individual parameters, database objects, waveform objects, and recipe objects. The most common parameters are character-strings describing the station characteristics. Table 4 lists common parameters included in *initsite* objects and their descriptions.

TABLE 4: PARAMETERS STORED IN INITSITE

Parameter Name	Type	Description
<i>sta</i>	character-string	station code
<i>NetType</i>	character-string	network type (single station or array)
<i>StaType</i>	character-string	station type (single or 3-C)
<i>WaveType</i>	character-string	wave type (S/H/I)
<i>actual-start-time</i>	real	start-time used by application as opposed to start-time specified by user
<i>actual-end-time</i>	real	end-time used by application as opposed to end-time specified by user

Objects and containers of database-derived information are also stored in *initsite*. They contain information pertaining to the site, including sensor, station, or arrival-related data. Table 5 lists the most common of these items found in the *initsite* object. (See “Querying the Database” on page 41 for more information about *dbobjects*.)

TABLE 5: DATABASE OBJECTS STORED IN INITSITE

Object Name	Type	Description
<i>dbsite</i>	GObj	<b>site</b> table
<i>dbassoc</i>	GObj	<b>assoc</b> table
<i>dborigin</i>	GObj	<b>origin</b> table
<i>arsoc</i>	GObj	derived from <b>arrival</b> , <b>assoc</b> and <b>origin</b> tables

The most common items in the *initsite* object are recipe objects. A recipe object contains processing and control parameters for performing specific tasks. The recipes are generally set in the CVAR datastore by the *DFX/libutil* function `set-cvar-from-file!`. The file is the application's site par file (named *DFX-site-*

## ▼ Shared Functionality

`<application>.par`). This file will specify configuration files that contain the recipe parameters. Recipe objects have default parameter values; only parameters that differ from the default values need to be specified in the recipe files. Because the recipes are set in the CVAR datastore, they are cleared before being set if the *initsite* object is being created inside a loop over stations. If this is not done, parameters meant for the previous station will be used instead of the default values intended by the user. The recipe values are cleared from the CVAR using the global DFX Scheme functions `delete-<recipe_name>-recipe-cvar`. Table 6 lists the recipes used among the current DFX applications.

**TABLE 6: RECIPES STORED IN INITSITE**

Recipe Name	Type	Description
<i>qc-rec</i>	GObj	parameters for controlling the quality of data
<i>tirec-con</i>	container	parameters for computing time intervals
<i>beamrec-con</i>	container	parameters for steering and filtering beams and setting detection thresholds
<i>det-rec</i>	GObj	detector parameters
<i>detbeamrec-con</i>	container	(see <i>beamrec-con</i> )
<i>amprec-con</i>	container	amplitude estimation parameters
<i>onset-rec</i>	GObj	onset estimation parameters
<i>SNRfilt-rec</i>	GObj	parameters for filtering waveforms for refined auto-regressive onset time estimation
<i>beamtirec-con</i>	container	corresponding <i>beamrec</i> and <i>tirec</i> objects used to create detection beams
<i>fk-rec</i>	GObj	parameters for f-k spectral analysis
<i>polar-rec</i>	GObj	parameters for measuring polarization at 3-C stations

**TABLE 6: RECIPES STORED IN INITSITE (CONTINUED)**

Recipe Name	Type	Description
<i>amp3crec-con</i>	container	parameters for 3-C amplitude computations
<i>tm-rec</i>	GObj	parameters for Threshold Monitoring
<i>smult-rec</i>	GObj	parameters for computing spectral variance and cepstral peaks (source multiplicity analysis)
<i>splp-rec</i>	GObj	parameters for computing short-period to long-period ratios in event characterization
<i>complexity-rec</i>	GObj	parameters for estimating complexity of signal in event characterization
<i>tf-rec</i>	GObj	parameters for performing time-frequency analysis in event characterization
<i>hydro-rec</i>	GObj	parameters for computing hydroacoustic features

Waveform objects may also be stored in the *initsite* object. The waveform objects are generally initialized when the *initsite* is created, but the actual waveforms are not read until later in the application; they are listed in Table 7. For more information about waveform objects see “Reading Waveforms” on page 58.

**TABLE 7: WAVEFORM OBJECTS STORED IN INITSITE OBJECTS**

Object Name	Type	Description
<i>wfdata</i>	GObj	<b>site</b> and <b>wfdisc</b> information
<i>wfmem</i>	GObj	waveforms
<i>stdbeam</i>	GObj	contains standard beams

Table 8 associates the objects found in *initsite* with each *DFX* application that employs the object class.

## ▼ Shared Functionality

TABLE 8: OBJECTS IN INITSITE PER APPLICATION

		DFX Application												
Object Name		SeisDet	OriginBeam	BOTF	IntSeisRcl	AutoSeisRcl	DPSNR	NoiseAmp	SeisEvch	HydroDet	IntHydroRcl	HydroEvch	SegArch	
Parameters	sta	X			X	X	X			X	X			
	NetType	X			X	X			X	X	X			
	StaType	X			X	X			X	X	X			
	WaveType	X			X	X				X	X			
	actual-start-time	X			X	X				X	X			
	actual-end-time	X			X	X				X	X			
	BandType1								X					
	BandType3								X					
Database Objects	dbsite		X	X	X	X		X	X			X	X	
	dbassoc					X								
	dborigin				X	X								
	arsoc							X	X				X	
Waveform Objects	wfdata	X	X	X	X	X	X	X	X	X	X	X	X	
	wfmem		X	X				X	X			X	X	
	stdbeam	X			X	X				X				

TABLE 8: OBJECTS IN INITSITE PER APPLICATION (CONTINUED)

	Object Name	DFX Application	SeisDet	OriginBeam	BOTF	IntSeisRcl	AutoSeisRcl	DPSNR	NoiseAmp	SeisEvch	HydroDet	IntHydroRcl	HydroEvch	SegArch
Recipes	<i>qc-rec</i>		X	X	X	X	X	X	X	X	X	X	X	X
	<i>tirec-con</i>		X	X	X	X	X	X	X	X			X	X
	<i>beamrec-con</i>		X	X	X	X	X	X	X	X	X		X	X
	<i>det-rec</i>		X			X	X	X			X	X		
	<i>detbeamrec-con</i>					X								
	<i>amprec-con</i>		X			X	X	X	X	X			X	X
	<i>onset-rec</i>		X								X			
	<i>SNRfilt-rec</i>		X											
	<i>beamtirec-con</i>		X											
	<i>fk-rec</i>		X			X	X							
	<i>polar-rec</i>		X			X	X							
	<i>amp3crec-con</i>		X			X	X							
	<i>tm-rec</i>		X											
	<i>smult-rec</i>								X	X				
	<i>splp-rec</i>									X				
	<i>complexity-rec</i>									X				
	<i>tf-rec</i>									X				
	<i>hydro-rec</i>										X	X		

## ▼ Shared Functionality

**Creating Initsite Objects**

Two general methods are used to initialize site information. The method used depends on whether the application is station or event based. The applications that operate on stations (such as detection processing) initialize a single station at a time, whereas applications that operate on a given set of events initialize a network of stations. Site initialization consists of creating *initsite* objects. Each station has its own *initsite* object and, therefore, applications that initialize a network of stations create an *initsite* container. Table 9 indicates which method is used by each of the applications.

**TABLE 9: METHOD USED TO CREATE INITSITE**

Algorithm	DFX Application													
	SeisDet	OriginBeam	BOTF	IntSeisRcl	AutoSeisRcl	DPSNR	NoiseAmp	SeisEvch	HydroDet	IntHydroRcl	HydroEvch	InfraDet	SegArch	QCStats
single station	X			X	X	X			X	X				
network of stations		X					X	X			X			
variation			X										X	
initsite not used												X		X

**Single Station Initialization**

The single station approach is straightforward. The application calls an internal function named `create-<application>-initsite`. This function takes the station code as input (there can be additional input arguments depending on the application) and returns an *initsite* object. Figure 7 shows a representative flow diagram.



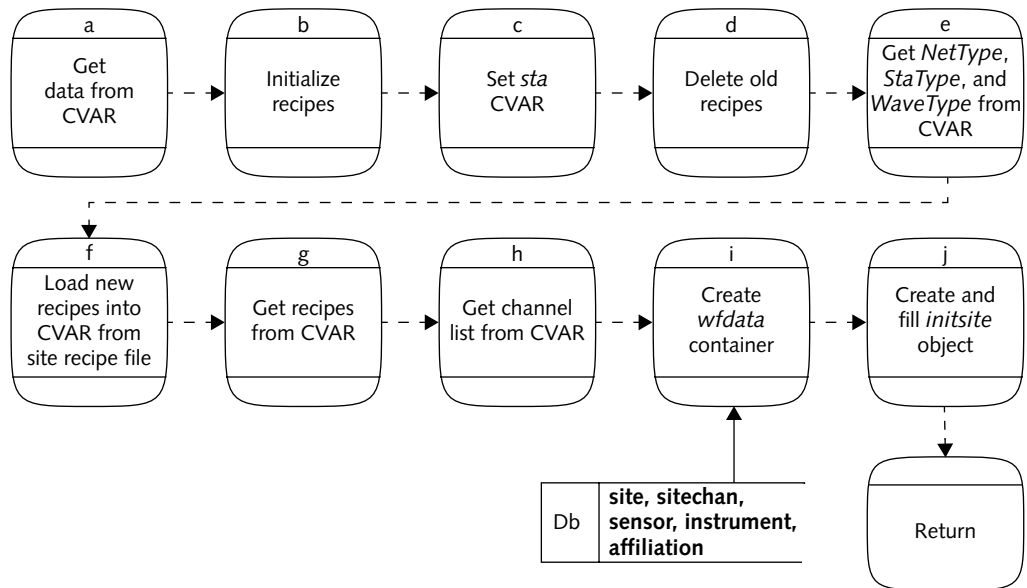


FIGURE 7. ARCHITECTURE OF CREATE-INITSITE

The standard algorithm for this site initialization method is as follows:

- a. Get site configuration filename from the CVAR datastore.  
Get table names from the CVAR datastore for **affiliation**, **site**, **sitechan**, **sensor**, and **instrument** tables.
- b. Create local recipe object symbols.
- c. Set station par in the CVAR datastore (`set-cvar! "sta" sta`).
- d. Delete old recipes from the CVAR datastore.
- e. Get *NetType*, *StaType*, and *WaveType* from the CVAR datastore.
- f. Load recipes for station into CVAR datastore using parameters constructed from the station name and the site configuration file (`set-cvar-from-file!`).
- g. Fill recipe objects from the CVAR datastore.
- h. Get list of channels from CVAR.

## ▼ Shared Functionality

- i. Create *wfdata* container via `query-for-wfdata-by-sta-chan-time`.
- j. Create and fill *initsite* object with station code, recipes, *NetType*, *StaType*, *WaveType*, and *wfdata* container.

**Network Initialization**

The event-driven *DFX* applications that initialize a network of stations use a different algorithm than single-station initialization. The network name is obtained from the CVAR datastore by the parameter *net* and corresponds to an **affiliation** table network. The application initializes all of the stations belonging to the network by the local Scheme function `initialize-net-sites`, which takes an *origin* container as input and returns an *initsite* container. Figure 8 shows a representative flow diagram.

The standard algorithm for this network initialization method is as follows:

- a. Get network and table names from CVAR.  
Get the list of stations from the **affiliation** table.
- b. Create empty *dbsite*, *initsite*, *wfdata* and *wfmem* containers.
- c. Fill *dbsite* container via `query-for-dbsite-by-<xxx>`  
(see “Querying the Database” on page 41).
- d. Fill *initsite* container via  
`create-initsite-container-from-dbsite-container`.
- e. Initialize travel-time tables via `initialize-net-tt-tables`  
(see “Determining Data Time Intervals” on page 64).
- f. Estimate time interval needed via `compute-net-ti`  
(see “Reading Waveforms” on page 58).
- g. Create and fill *wfdata* container via  
`query-for-wfdata-by-sta-chan-time`  
(see “Querying the Database” on page 41 and “Reading Waveforms” on page 58).

- h. Create and fill *wfmem* container via *query-for-wfmem-by-sta-chan-time* (see “Querying the Database” on page 41 and “Reading Waveforms” on page 58).
- i. Insert *wfdata* and *wfmem* containers into *initsite* container via *fill-initsite-container*.

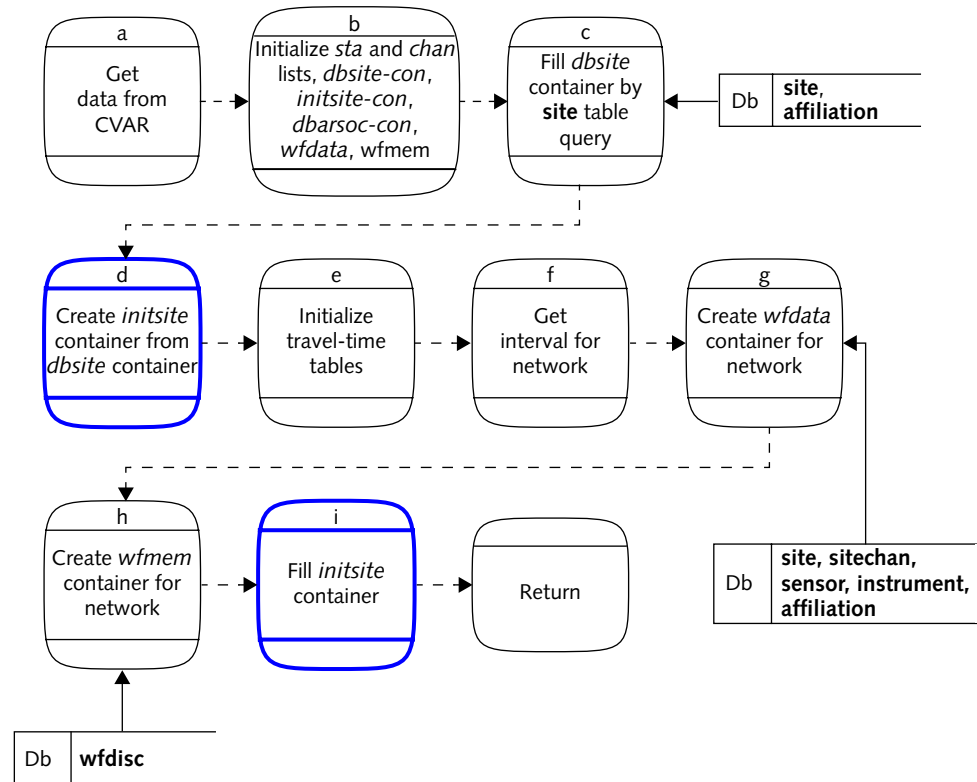


FIGURE 8. ARCHITECTURE OF INITIALIZE-NET-SITES

The *initsite* container is created using the set of sites in the *dbsite* container by the local Scheme function *create-initsite-container-from-dbsite-container*. Figure 9 shows a flow diagram of this function. The function loops over the *dbsite* container.

## ▼ Shared Functionality

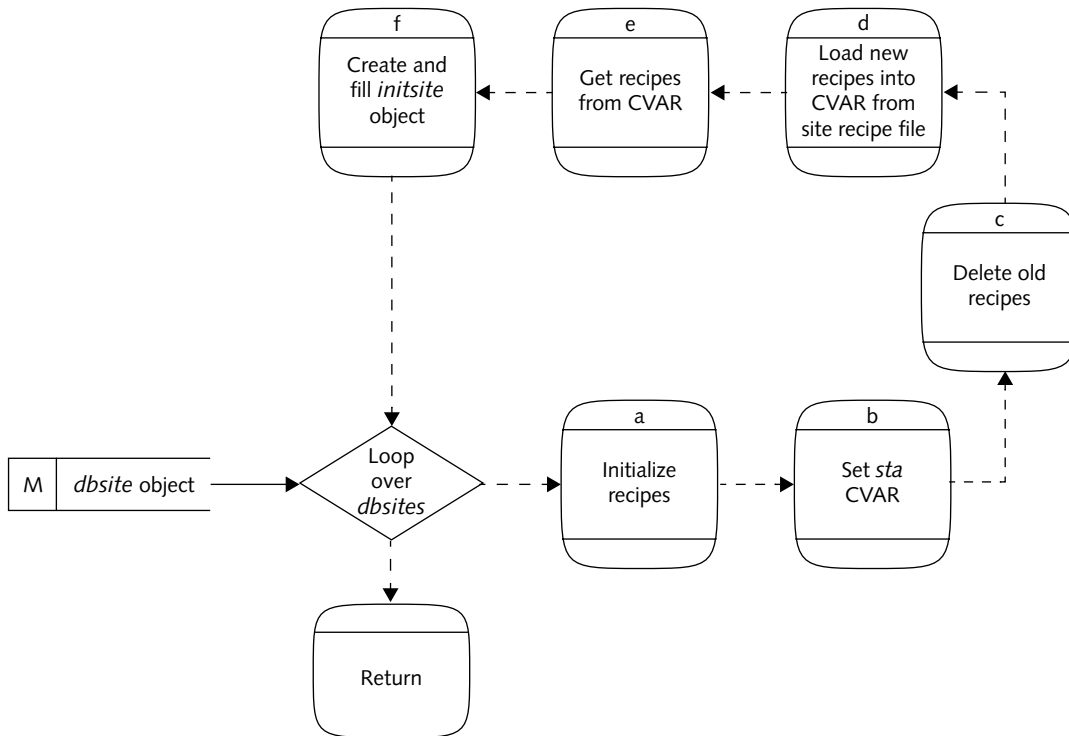


FIGURE 9. ARCHITECTURE OF CREATE-INITSITE-CONTAINER-FROM-DBSITE-CONTAINER

The following algorithm is performed for each *dbsite* object:

- a. Create local recipe object symbols.
- b. Set station name parameter *sta* in the CVAR datastore.
- c. Delete old recipes from the CVAR datastore.
- d. Load recipes for station into CVAR datastore using parameters constructed from the station name and the site configuration file (*set-cvar-from-file!*).
- e. Fill recipe objects from the CVAR datastore.
- f. Create empty *initsite* object and fill with *dbsite* object and desired recipes.

The *wfdata* and *wfmem* containers must be inserted into the appropriate *initsite* objects. This is accomplished by the local DFX Scheme function `fill-initsite-container`. Figure 10 is a flow diagram of this function. The function loops over the *initsite* container, and the following algorithm is carried out for each *initsite*:

- a. Retrieve the station code *sta* from the *initsite*'s *dbsite* object attribute.
- b. Create a station-specific *wfdata* container from the station objects in the original *wfdata* container.
- c. Remove any duplicated *wfdata* objects from the newly created container.
- d. Create a station-specific *wfmem* container from the station objects in the original *wfmem* container.
- e. Insert station *wfdata* and *wfmem* objects into the current *initsite* object.

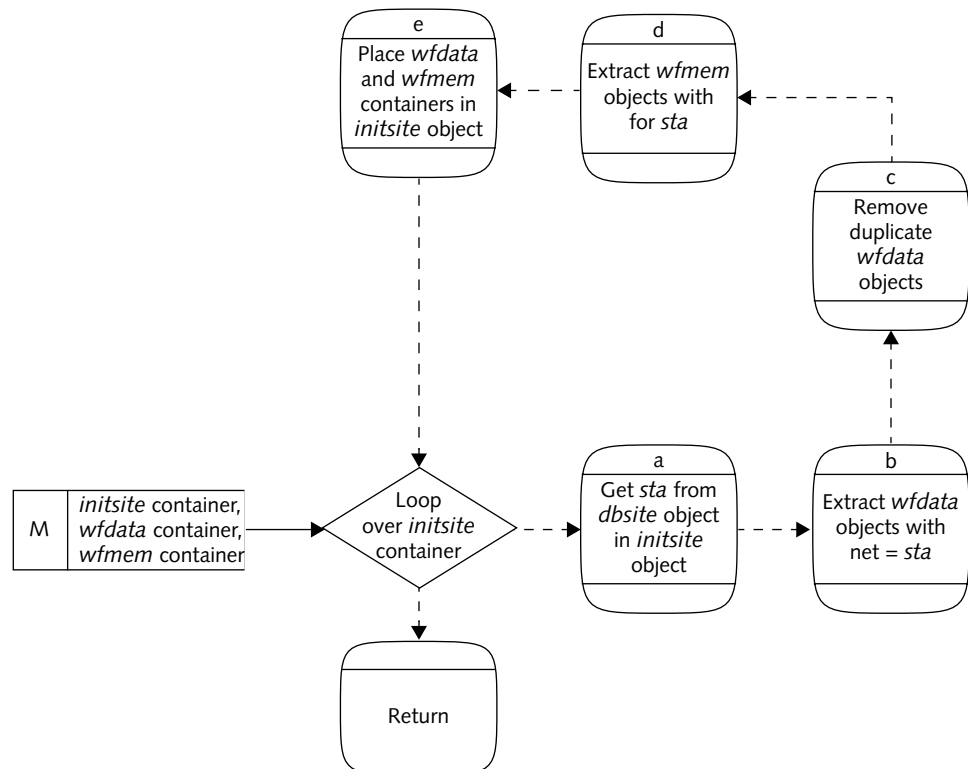


FIGURE 10. ARCHITECTURE OF FILL-INITSITE-CONTAINER

## ▼ Shared Functionality

## READING WAVEFORMS

Most *DFX* applications operate on waveforms. A standard method of reading and managing waveforms has been developed. There are two classes of objects for handling waveforms: *wfdata* and *wfmem*. The *wfdata* and *wfmem* objects must be created and filled before reading the actual waveforms. After the waveforms are read, the *wfdata* objects contain a pointer to the *wfmem* objects where the actual time series data are stored. The *wfdata* and *wfmem* objects are described in more detail in the following sections.

### Waveform Objects

#### *wfdata* objects

The *wfdata* object stores information from the **site**, **sitechan**, **sensor**, **instrument**, and **affiliation** tables. It also contains pointers to a *wfmem* object. The *wfdata* container is created and filled by the global *DFX* Scheme function `query-for-wfdata-by-sta-chan-time`. The arguments of this function are listed in Table 10. The exact start- and end-times of the processing interval are not critical here because the tables that are queried by this function use the time values only to identify dates of station and instrument modification. The function returns a container that has a *wfdata* object for each station/channel pair. Most *DFX* applications create and store *wfdata* containers in the *initsite* object.

**TABLE 10: FUNCTIONAL INTERFACE OF QUERY-FOR-WFDATA-BY-STA-CHAN-TIME**

Argument Name	Usage	Type	Description
<i>sta-list</i>	input	list of strings	station codes
<i>chan-list</i>	input	list of strings	channel codes
<i>start-time</i>	input	real	epoch start-time of desired interval
<i>end-time</i>	input	real	epoch end-time of desired interval

**TABLE 10: FUNCTIONAL INTERFACE OF QUERY-FOR-WFDATA-BY-STA-CHAN-TIME (CONTINUED)**

Argument Name	Usage	Type	Description
<i>site-table</i>	input	character-string	<b>site</b> table name
<i>sitechan-table</i>	input	character-string	<b>sitechan</b> table name
<i>sensor-table</i>	input	character-string	<b>sensor</b> table name
<i>instrument-table</i>	input	character-string	<b>instrument</b> table name
<i>affiliation-table</i>	input	character-string	<b>affiliation</b> table name
n/a	returned value	container	<i>wfdata</i> objects

The *wfdata* object contains information about the site and the instrument. Table 11 lists the contents of a *wfdata* object.

**TABLE 11: CONTENTS OF WFDATA OBJECT**

Attribute Name	Type	Source Table	Name of Database Attribute	Description
<i>sta</i>	character-string	<b>site</b>	<i>sta</i>	station code
<i>chan</i>	character-string	<b>site</b>	<i>chan</i>	channel code
<i>staondate</i>	integer	<b>site</b>	<i>ondate</i>	first day this record is valid
<i>staoffdate</i>	integer	<b>site</b>	<i>offdate</i>	last day this record is valid
<i>lat</i>	real	<b>site</b>	<i>lat</i>	latitude of site (deg)
<i>lon</i>	real	<b>site</b>	<i>lon</i>	longitude of site (deg)
<i>elev</i>	real	<b>site</b>	<i>elev</i>	elevation of site (m)
<i>staname</i>	character-string	<b>site</b>	<i>staname</i>	station name
<i>statype</i>	character-string	<b>site</b>	<i>statype</i>	station type
<i>refsta</i>	character-string	<b>site</b>	<i>refsta</i>	reference station code

## ▼ Shared Functionality

TABLE 11: CONTENTS OF WFDATA OBJECT (CONTINUED)

Attribute Name	Type	Source Table	Name of Database Attribute	Description
<i>dnorth</i>	real	<b>site</b>	<i>dnorth</i>	offset in north-south direction from reference station (km)
<i>deast</i>	real	<b>site</b>	<i>deast</i>	offset in east-west direction from reference station (km)
<i>chanondate</i>	integer	<b>sitechan</b>	<i>ondate</i>	start day channel information is valid
<i>chanoffdate</i>	integer	<b>sitechan</b>	<i>offdate</i>	end day channel information is valid
<i>chanid</i>	integer	<b>sitechan</b>	<i>chanid</i>	channel identifier number
<i>ctype</i>	character-string	<b>sitechan</b>	<i>ctype</i>	channel type
<i>edepth</i>	real	<b>sitechan</b>	<i>edepth</i>	emplacement depth (m)
<i>vang</i>	real	<b>sitechan</b>	<i>vang</i>	sensor orientation angle from vertical (deg)
<i>hang</i>	real	<b>sitechan</b>	<i>hang</i>	sensor orientation angle clockwise from north (deg)
<i>chandescr</i>	character-string	<b>sitechan</b>	<i>descrip</i>	channel description
<i>senstime</i>	real	<b>sensor</b>	<i>time</i>	begin time of <b>sensor</b> record
<i>sensendtime</i>	real	<b>sensor</b>	<i>endtime</i>	end-time of <b>sensor</b> record
<i>inid</i>	integer	<b>sensor</b>	<i>inid</i>	instrument identifier
<i>sensjdate</i>	integer	<b>sensor</b>	<i>jdate</i>	date of <b>sensor</b> record
<i>calratio</i>	real	<b>sensor</b>	<i>calratio</i>	calibration update factor
<i>calper</i>	real	<b>sensor</b>	<i>calper</i>	period for calibration data
<i>tshift</i>	real	<b>sensor</b>	<i>tshift</i>	data time correction
<i>instant</i>	character-string	<b>sensor</b>	<i>instant</i>	discrete or continuous flag
<i>insname</i>	character-string	<b>instrument</b>	<i>insname</i>	instrument name



TABLE 11: CONTENTS OF WFDATA OBJECT (CONTINUED)

Attribute Name	Type	Source Table	Name of Database Attribute	Description
<i>instype</i>	character-string	<b>instrument</b>	<i>instype</i>	instrument type
<i>band</i>	character-string	<b>instrument</b>	<i>band</i>	frequency band
<i>digital</i>	character-string	<b>instrument</b>	<i>digital</i>	data type
<i>samprate</i>	real	<b>instrument</b>	<i>samprate</i>	nominal sampling rate (samples/s)
<i>ncalib</i>	real	<b>instrument</b>	<i>ncalib</i>	nominal calibration factor
<i>ncalper</i>	real	<b>instrument</b>	<i>ncalper</i>	nominal calibration period
<i>insdir</i>	character-string	<b>instrument</b>	<i>dir</i>	directory containing instrument response
<i>insdfile</i>	character-string	<b>instrument</b>	<i>dfile</i>	file containing instrument response
<i>rsptype</i>	character-string	<b>instrument</b>	<i>rsptype</i>	response type
<i>wfmem</i>	pointer to GObj	n/a	n/a	points to waveform memory object
<i>mask</i>	pointer to GObj	n/a	n/a	points to quality-control mask object
<i>net</i>	character-string	<b>affiliation</b>	<i>net</i>	network affiliation

### wfmem Objects

The *wfmem* object contains the **wfdisc** table information and the time series data. The *wfmem* container is created and filled from the **wfdisc** table by the global DFX Scheme function `query-for-wfmem-by-sta-chan-time`

The arguments of this function are listed in Table 12. The function returns a container that has *wfmem* objects that cover the time period requested for each station/channel pair. At this stage the *wfmem* objects do not contain the waveforms. The *wfmem* objects are either created and stored in the *initsite* object (see “Initial-

## ▼ Shared Functionality

izing Sites" on page 46) or created just before reading the waveforms. Event-driven applications generally perform the former, and station driven applications perform the latter.

**TABLE 12: FUNCTIONAL INTERFACE OF QUERY-FOR-WFMEM-BY-STA-CHAN-TIME**

Argument Name	Usage	Type	Description
<i>sta-list</i>	input	list of strings	list of station codes
<i>chan-list</i>	input	list of strings	list of channel codes
<i>start-time</i>	input	real	epoch start-time of desired interval
<i>end-time</i>	input	real	epoch end-time of desired interval
<i>wfdisc-table</i>	input	character-string	<b>wfdisc</b> table name
<i>ex-time</i>	input	real	time extension to ensure that all wfdiscs are retrieved
n/a	returned value	container	<i>wfmem</i> objects

The start-time and end-time for this function are important because the *time* and *endtime* attributes are part of the **wfdisc**'s primary key. However, it is better for the time window to be too large rather than too small to ensure that the **wfdisc** records will cover the actual time interval desired. For event-driven processing, the time interval is found by estimating the minimum and maximum arrival times for all origins and stations. This is why all event *wfmems* are read upfront. The time intervals are determined by computing travel times between all origins and stations by the local Scheme function `compute-net-t.i`. An extra time segment is added to both sides of the time interval to ensure that enough data are available for processing near the interval's edges. Station-driven processing uses the *DFX/libdata* function `get-data-interval` ("Determining Data Time Intervals" on page 64 for more information on time interval determination).

The *wfmem* object contains the **wfdisc** table information and, once they are read, the waveforms. Table 13 lists the contents of a *wfmem* object.

TABLE 13: CONTENTS OF WFMEM OBJECT

Attribute	Type	Source Table	Name of Database Attribute	Description
<i>sta</i>	string	<b>wfdisc</b>	<i>sta</i>	station code
<i>chan</i>	character-string	<b>wfdisc</b>	<i>chan</i>	channel code
<i>time</i>	real	<b>wfdisc</b>	<i>time</i>	begin time of <b>wfdisc</b> record
<i>wfid</i>	integer	<b>wfdisc</b>	<i>wfid</i>	waveform identifier
<i>chanid</i>	integer	<b>wfdisc</b>	<i>chanid</i>	channel identifier
<i>jdate</i>	integer	<b>wfdisc</b>	<i>jdate</i>	date of <b>wfdisc</b> record
<i>endtime</i>	real	<b>wfdisc</b>	<i>endtime</i>	end-time of <b>wfdisc</b> record
<i>nsamp</i>	integer	<b>wfdisc</b>	<i>nsamp</i>	number of samples for <b>wfdisc</b>
<i>samprate</i>	real	<b>wfdisc</b>	<i>samprate</i>	sample rate (samples/s)
<i>calib</i>	real	<b>wfdisc</b>	<i>calib</i>	calibration factor
<i>calper</i>	real	<b>wfdisc</b>	<i>calper</i>	calibration period
<i>instype</i>	string	<b>wfdisc</b>	<i>instype</i>	instrument type
<i>segtype</i>	character-string	<b>wfdisc</b>	<i>segtype</i>	segment type
<i>datatype</i>	character-string	<b>wfdisc</b>	<i>datatype</i>	data type
<i>clip</i>	character-string	<b>wfdisc</b>	<i>clip</i>	data clipped flag
<i>dir</i>	character-string	<b>wfdisc</b>	<i>dir</i>	directory of waveforms file
<i>dfile</i>	character-string	<b>wfdisc</b>	<i>dfile</i>	waveform filename
<i>foff</i>	integer	<b>wfdisc</b>	<i>foff</i>	data file byte offset
<i>commid</i>	integer	<b>wfdisc</b>	<i>commid</i>	comment identifier
<i>lddate</i>	character-string	<b>wfdisc</b>	<i>lddate</i>	date wfdisc record created
<i>data</i>	pointer to time-series	n/a	n/a	points to data array

## ▼ Shared Functionality

**Determining Data Time Intervals**

Before reading waveforms, the desired time interval must be determined. The method used to find the time interval depends on whether the application is station driven or event driven.

**Event Processing**

For event-driven processing, a rough estimate of the interval is needed to create the *wfmem*, but the interval chosen is generally large enough to encompass the entire set of origins and sites. A much smaller time window (potentially) is needed for a single origin/station pair. The time interval is determined by the *DFX/libutil* function `compute-site-origin-ti`. The arguments for this function are listed in Table 14. The function returns a start-time and an end-time (in a list) that encompass the predicted arrival time of the phase as specified by the time interval recipe. An extra lead and lag time are added to account for filter edge effects, beamforming lags, and other data processing issues.

**TABLE 14: FUNCTIONAL INTERFACE OF COMPUTE-SITE-ORIGIN-TI**

Argument Name	Usage	Type	Description
<i>site</i>	input	GObj	<i>initsite</i> object
<i>origin</i>	input	GObj	<i>dborigin</i> object
<i>tirec-con</i>	input	GObj	time interval recipes
n/a	returned value	list of reals	computed start-time and end-time

Prior to calling `compute-site-origin-ti` the travel-time tables must be initialized. An interface to the common library *libloc* has been developed for use in *DFX*. The travel-time tables are initialized with the *DFX/libutil* function `read-default-travel-time-tables`. This function is called by the Scheme language call:

```
(read-default-travel-time-tables dbsite-con phase-list)
```

The travel-time tables for the phases defined in *phase-list* and the stations in the *dbsite* container are read into memory and stored. They can be accessed later by other *DFX/libutil* functions.

### Station Processing

In station processing the data and processing time intervals are determined by the *DFX/libdata* functions *get-data-interval* and *get-processing-interval*, respectively. The data interval is the entire waveform time segment that is retrieved. The processing interval is the portion within the data interval that is used to find detections. The excess data allows for computations such as the STA and LTA time series to be initialized, and it assures that filter edge effects do not create artifacts in detection processing. The actual time intervals depend on the data that are available.

The function *get-data-interval* takes the three input arguments listed in Table 15. The returned time interval extends the user-specified times. The extended amount is determined from the parameters *data-ltav-stability-fraction*, *det-ltav-len*, and *data-offset-len*. These parameters are obtained from recipe files via the CVAR datastore. A time window known as the “LTA stability length” is used to compute the time extension amount and is defined as:

$$\text{data-ltav-stability-len} = \text{data-ltav-stability-fraction} * \text{det-ltav-len}.$$

The time extension on either side of the interval is:

$$\text{time-extension} = 2 * \text{data-offset-len} + \text{data-ltav-stability-len}.$$

The returned interval is passed to *read-waveforms* which acquires the waveforms that are available for this interval (see “Reading Binary Files” on page 69). The extent of the available data defines the actual data interval. This is used by *get-processing-interval* to determine the final processing interval.

## ▼ Shared Functionality

TABLE 15: FUNCTIONAL INTERFACE OF GET-DATA-INTERVAL

Argument Name	Usage	Type	Description
<i>det-rec</i>	input	GObj	detection recipe
<i>start-time</i>	input	real	user specified epoch start-time
<i>end-time</i>	input	real	user specified epoch end-time
n/a	returned value	list of reals	computed start-time and end-time

The function `get-processing-interval` takes the four input arguments listed in Table 16. It is invoked after the waveforms are read by `read-waveforms` (see “Reading Binary Files” on page 69). The returned time interval begins *data-ltav-stability-len* + *data-offset-len* seconds after the start of the data and terminates *data-offset-len* seconds before the end of the data.

TABLE 16: FUNCTIONAL INTERFACE OF GET-PROCESSING-INTERVAL

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	GObj	waveforms
<i>det-rec</i>	input	GObj	detection recipe
<i>start-time</i>	input	real	user specified epoch start-time
<i>end-time</i>	input	real	user specified epoch end-time
n/a	returned value	list of reals	computed processing start-time and end-time

The processing interval computed from the above steps differs from the user requested time interval. The degree to which it differs depends on the data availability. There are four cases to consider as shown in Figure 11. In the first case, (a), data are available for the entire extended data interval. This can occur when station processing is running behind the data acquisition or when DFX is run offline. In this case, the actual processing interval is larger than the original requested interval. In case (b), data are not available after the requested time interval. This

can occur when DFX is running near real-time (typical under normal operational conditions). The final interval in this case is the same length as the requested interval but shifted by *data-offset-len* seconds. In case (c), data are available for the requested time and at later times, but data are not available prior to the requested interval. This can occur when data do not arrive sequentially. The final interval in this case is also the same length as the requested time interval as it was in case (b), but it is shifted in the opposite direction by *data-offset-len* plus *data-ltav-stability-len* seconds. The last case, (d), is an isolated data segment. The final time interval is smaller than the requested interval. It starts *data-offset-len* plus *data-ltav-stability-len* seconds beyond where the data begins, and terminates *data-offset-len* seconds prior to the end of the data. Because of these variations and the effect they have on screening, results from DFX processing offline can differ slightly from DFX processing in operations.

## ▼ Shared Functionality

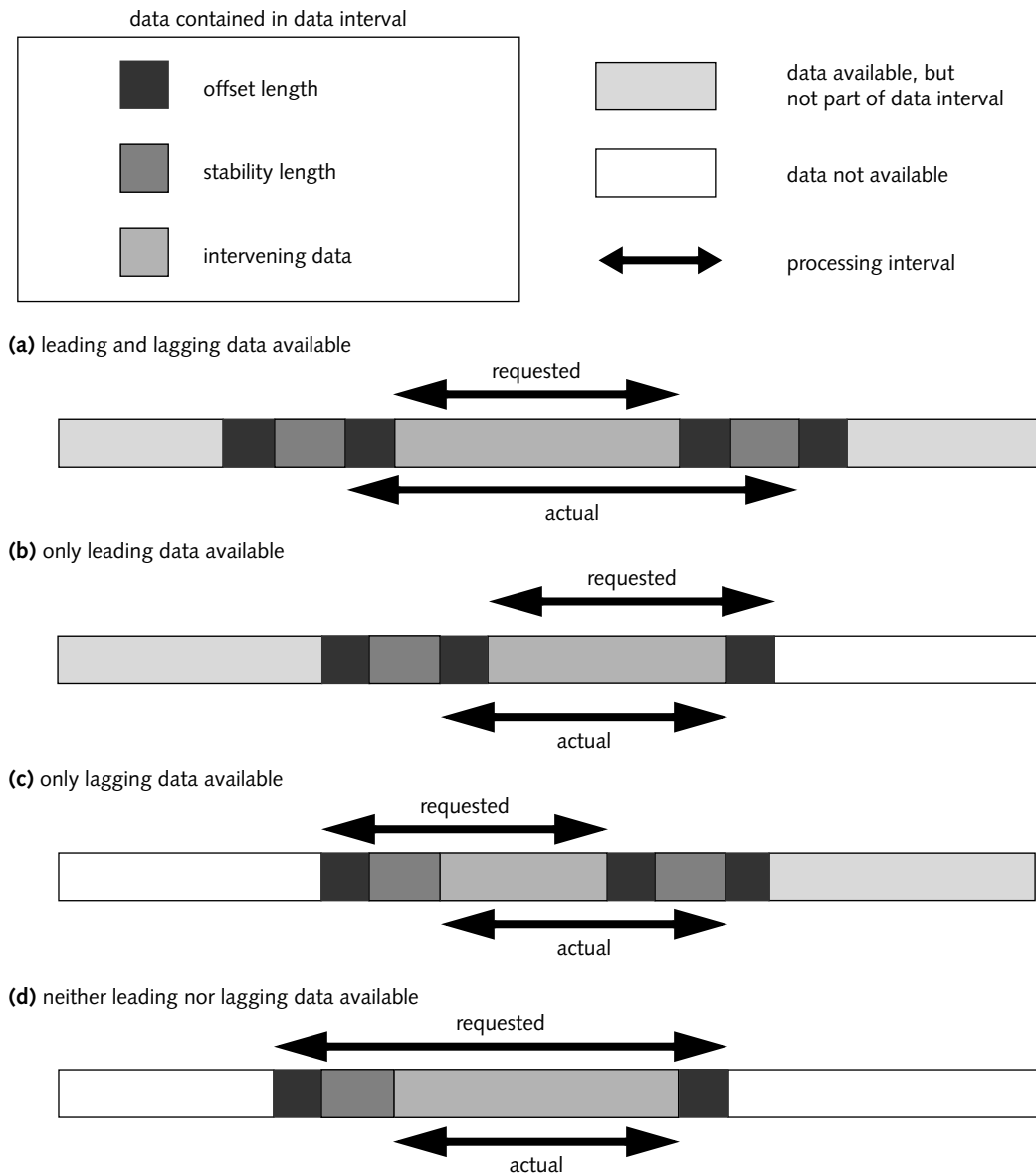


FIGURE 11. COMPUTED DATA AND PROCESSING TIME INTERVALS



## Reading Binary Files

The waveforms are read by the *DFX/libdata* function `read-waveforms`. This function is called by the Scheme language call:

```
(read-waveforms wfmem-con wfdata-con desired-start-time
  desired-end-time)
```

The arguments for the function are listed in Table 17.

**TABLE 17: FUNCTIONAL INTERFACE OF READ-WAVEFORMS**

Argument Name	Usage	Type	Description
<i>wfmem-con</i>	input & output	container	<i>wfmem</i> objects
<i>wfdata-con</i>	input & output	container	<i>wfdata</i> objects
<i>desired-start-time</i>	input	real	requested start-time of the data interval
<i>desired-end-time</i>	input	real	requested end-time of the data interval

The input *wfdata* container may have multiple objects for a given station/channel pair, one for each *wfdisc* record that lies within the time interval. The function `read-waveforms` will replace the multiple *wfmem* objects with a single object containing the waveforms for the particular station/channel. The modified *wfmem* container is set into the *wfdata*'s *wfmem* attribute.

The waveforms that are returned by `read-waveforms` have had the calibration information in the *wfdata* objects applied to them. The waveforms are zero padded, if needed, so that all waveforms start and end at the same time.

## Checking Quality of Waveforms

After the waveforms are read, most *DFX* applications check the quality of the data. The purpose of this process is to find and fix or remove defective data samples and to discard bad data channels so that they will not degrade the signal processing results. The data quality is determined by a complex algorithm that records the

## ▼ Shared Functionality

location of the defective data in “masks,” which are saved as objects within the *wfdata* objects. Waveforms that have large masked sections are generally dropped from further processing by deleting the objects from the waveform container. The routines that check the quality of the data can also perform some data repairs. The most common of these routines is the *demean* option, which removes the DC offset.

In *DFX* applications, the data quality check is accomplished by the *DFX/libqc* functions *qc-waveforms* and *qc-waveforms-all*. These functions are called with the Scheme language calls:

```
(qc-waveforms wfdata-con qc-rec)
(qc-waveforms-all wfdata-con qc-rec)
```

Both functions take a *wfdata* container and a quality-control recipe object as input (Table 18). The only difference between the two functions is that *qc-waveforms-all* returns all of the waveforms, whereas *qc-waveforms* deletes waveforms that are determined to have a large fraction of bad data as identified by the recipe parameter *qc-max-mask-fraction*.

**TABLE 18: FUNCTIONAL INTERFACE OF QC-WAVEFORMS AND QC-WAVEFORMS-ALL**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input and output	container	<i>waveform</i> objects
<i>qc-rec</i>	input	GObj	quality-control parameters
n/a	returned value	container	<i>waveform</i> objects

## COMMON DFX SCHEME TASKS

### Defining Generic Objects

Generic objects are used throughout *DFX* for holding and transferring most of the data that are manipulated in the Scheme applications. These generic objects are often referred to throughout this document as “objects” or “GObjs.” For a general description of generic objects see “*DFX Data Representations*” on page 18.

Generic objects store various types of data such as database records, detection attributes, and recipe parameters. Generic objects are created by the *libgobj* Scheme function `create-gobj`. This function takes two character-strings as input and returns a pointer to the object. The first character-string is used to name this instantiation of the object. The second character-string is the class name of the generic object and corresponds to the class definitions created in the C libraries or with the Scheme function `define-gobj-class`. Instead of entering a character-string directly for the class name, a set of *DFX* Scheme functions with the naming convention `say-<name>-object-class` are used where *<name>* is the name of the object class. The following example creates a detection object:

```
(let ((det (create-gobj
  (make-gobj-name "det" "obj") (say-det-object-class))))
```

The global *DFX* Scheme function `say-det-object-class` provides the class identifier of a detection-class object. The *DFX/libutil* function `make-gobj-name` creates an unique character-string that is used to identify the object internally (the user does not need to know what this character-string actually is). The function `create-gobj` creates the object in memory. The `let` command then binds the object to the symbol `det`, which is used to reference the object in subsequent processing. At this point the object has been instantiated, and its attributes contain only default values.

*DFX* has several standard object classes. One such class is the vector class, which is used for data that are best represented by vectors, such as time series or waveforms. Vector objects can be manipulated arithmetically. Another standard object class is the database-connection-object class, which is integrated with *libgdi* and is

## ▼ Shared Functionality

used to maintain a database connection instance. Multiple instances of database connections may be maintained this way. Instances of this class contain function attributes that are used to interact with the database.

Table 19 lists several common generic object classes available in *DFX*. Most of the objects are defined with *DFX* library functions; however, a few, such as the *initsite* object, are defined locally within the application.

**TABLE 19: COMMON GENERIC OBJECT TYPES**

Object Name	Scheme Function Used to Obtain Class Name	Description
<i>db&lt;table&gt;</i>	<code>say-db&lt;table&gt;-object-class</code>	objects that correspond to database tables (see "Database Operations" on page 40)
<i>dbconnection</i>	<code>say-dbconnection-object-class</code>	object used to establish database connection (see "Database Operations" on page 40)
<i>initsite</i>	<code>say-initsite-object-class</code>	object containing site -specific information; definition local to the application (see "Initializing Sites" on page 46)
<i>wfdata</i>	<code>say-wfdata-object-class</code>	object containing sensor information (see "Reading Waveforms" on page 58)
<i>wfmem</i>	<code>say-wfmem-object-class</code>	object containing waveforms (see "Reading Waveforms" on page 58)
<i>detection</i>	<code>say-det-object-class</code>	object containing detection attributes (distinct from <i>dbdetection</i> object)
<i>trigger</i>	<code>say-trig-object-class</code>	object containing trigger attributes
<i>beam</i>	<code>say-beam-object-class</code>	object containing beams
<i>beam-rec</i>	<code>say-beamrec-object-class</code>	object containing beam recipe parameters

The attributes of the generic objects are accessed by two functions: `set-gobj-attr!` and `extract-gobj-attr`. The functions are called by the following Scheme language calls:

```
(set-gobj-attr! <object> "<attribute_name>" <attribute_value>)
(let ((<var> (extract-gobj-attr <object> "<attribute_name>"))))
```

where `<object>` is the generic object that is being operated on. The attribute names are the same as defined in the object class definition.

The *beam* object is a common type of object that requires some explanation. The term "beam" has been generalized throughout the IDC system. A beam is essentially processed waveforms and does not necessarily refer to a "delay and sum" operation over several channels. Although *beam* objects are often used to hold traditional delay-and-sum beams, they are also used to hold filtered data. This allows the recipes and functions used to filter time-series data to have the same structure as the recipes and functions used in beamforming.

DFX applications often use multiple generic objects of the same class together in what is called a "container." The container is analogous to an array of structures in a C program. A container object maintains a table of the objects it contains that can be searched, expanded, partitioned, traversed, and so forth. The container is created in much the same way as the generic object, but uses the global DFX function `create-keyed-list-container` instead of `create-gobj`. The following example is a Scheme language call used to create a container:

```
(let ((<type>-con (create-keyed-list-container
  (make-gobj-name "<type>" "con") (say-<type>-object-class))))
```

The symbol `<type>-con` is bound to the container and is used to reference it in subsequent processing.

Many local and global DFX functions return objects or containers. In this case, the objects or containers do not need to be initialized as previously described. Instead the symbol name is created with an empty value by the Scheme call:

```
(let (var nil))
```

## ▼ Shared Functionality

An object returned by a function can then be bound to the symbol *var* using the `set!` command.

**Looping Over Contents of Containers**

*DFX* applications often process the objects in a container one at a time. This is accomplished using the global *DFX* Scheme function `map-container`. The following example runs the function `(function)` on each object in the container `object-con`.

```
(map-container function object-con)
```

**Internal Screening of Containers**

Objects within containers are checked against a set of criteria at various points within the application. The objects that fail to meet the criteria are removed from the container. This process is known as screening. Screening can be external or internal. External screening checks the objects in the container against duplicate objects in the database tables. External Screening is described in "Screening Database Containers" on page 43. For internal screening, the criteria differ depending on the object type and the application, but the types of criteria generally fall into two categories: screening-by-time and screening-by-attribute.

Screening-by-time removes detections that occur too close in time. The allowable time interval is set in the parameter files. Screening-by-attribute checks if the object's attributes fall within certain bounds. Those attributes that do not are removed from the container.

## Setting and Retrieving Parameters from CVAR Datastore

The CVAR datastore is used to load parameters and recipe tables into a *DFX* Scheme application. The parameters and recipe tables include database account names, input and output table names, and site-specific recipes. The CVAR datastore can consist of any number of parameters. Many are set upon the invocation of *DFX* from the input par file or from the command line directly.

CVAR datastore parameters can also be set from within the Scheme applications. Two functions are used to accomplish this: `set-cvar!` and `set-cvar-from-file!`. The function `set-cvar!` is used for setting a single parameter. It is called by the Scheme language call:

```
(set-cvar! "parameter_name" parameter_value)
```

Parameters that are defined in a file can be set with the function `set-cvar-from-file!`. This function is called by the Scheme language call:

```
(set-cvar-from-file! filename)
```

Most applications use this function to set the site-specific recipes when creating the *initsite* object (see “Initializing Sites” on page 46).

Parameters are retrieved from the CVAR datastore by four *libpar* functions: `getpar`, `getspar`, `mstpar`, and `mstspar`. The functions `getpar` and `mstpar` are used for numeric parameters; `getspar` and `mstspar` are used for character-string parameters. The functions `getpar` and `getspar` are used for optional parameters. That is, if the parameters do not exist, the functions use a default value. The functions `mstpar` and `mstspar` are used for required parameters and return an error if the parameters are not defined in the CVAR datastore (`mst` is short for “must”).

The recipe objects stored in the CVAR datastore are generally retrieved using functions with names similar to the name of the recipe. For example, the quality-control recipe is obtained using the function `init-qc-rec`.

## ▼ Shared Functionality

Recipe objects have default attribute values, which are used when the attributes are not specified in the configuration files. After a parameter is set in the CVAR datastore, it will remain there for the entire execution of the *DFX* run unless explicitly cleared. Many *DFX* applications loop over stations and retrieve recipes for each one. Because one station may set certain parameters while another uses the default values, it is important to delete any recipes that may be for previous stations. This is generally accomplished by the global *DFX* functions `delete-<name>-recipe-cvar`, where *<name>* is the name of the recipe.

**Error Handling**

Error handling in *DFX* Scheme applications is handled by the intrinsic `catch` and `throw-tag-error` functions. These functions are called as follows:

```
(*catch
  'error-tag-name
  (...the actual processing occurs here...
    (if (checks for error)
      (throw-tag-error
        'error-tag-name
        (string-format
          "Error message"
        )
      )
    )
  );; end of if block
);; end of processing block
);; end of catch block
;; on an error condition, processing control resumes here
```

The `catch` function defines an error tag that surrounds a block of code. The function `throw-tag-error` is placed inside the error check block, and if it is executed, causes the `catch` block that has the same tag as the first argument to `throw-tag-error` to return a value of *error*. The `throw-tag-error` function's second



argument is a character-string containing an error message that is printed prior to the throw. Nested catch blocks are allowed so that different error conditions can return control to different blocks.

*DFX* functions typically return nil or, when a function normally returns a container, they return an empty container. The following idiom is often used:

```
(if (container-empty? (do-something arg1 arg2))
    (throw-tag-error 'some-kind-of-error
                    "Error: cannot do-something"
                    )
    )
```

Here the function *do-something* operates on two arguments and is expected to return a container, but in this case the returned container is only used as an error check.

## Memory Management

Scheme handles the memory management of its programming environment through the use of a memory “heap” and a “garbage-collector.” The total memory allocated to Scheme is referred to as the heap. All variable lists created in the Scheme application are assigned a portion of the heap. The Scheme garbage-collector automatically identifies non-referenced memory and frees it for use by new entities. The default heap size is normally sufficient for routine processing, but it can be set explicitly by the *DFX* command line switch “-h” (see the *DFX* UNIX man page). The heap memory is strictly for Scheme constructs. Generic objects, the CVAR datastore, and the C library functions handle their own memory even though they may be bound to Scheme variables.



## Chapter 4: Seismic Processing

This chapter describes the design of *DFX* seismic processing and includes the following topics:

- Automatic Seismic Detection [1]
- Automatic Origin Beam [2]
- Interactive Beam-on-the-Fly [3]
- Interactive Seismic Recall [4]
- Automatic Seismic Recall [5]
- Automatic Depth-phase SNR [6]
- Automatic Noise Amplitude Estimation [7]
- Automatic Seismic Event Characterization [8]

## Chapter 4: Seismic Processing

### AUTOMATIC SEISMIC DETECTION [1]

*Automatic Seismic Detection (SeisDet)* is a DFX Scheme application that detects transient seismic signals and extracts features characterizing these detections. The source code resides in the file `DFX-detection.scm`. *SeisDet* is typically called in the automatic pipeline processing, but it can also be run offline. *SeisDet* is usually run by specifying the station, start-time, end-time, and the `DFX-detection.par` file on the command line. *SeisDet* makes detections using an STA/LTA type detector. For each detection, it computes a series of attributes of the detected signals specific to seismic processing. *SeisDet* also produces detection beams to support subsequent analyst review activities. The results of *SeisDet* are written to the **amplitude**, **amp3c**, **apma**, **arrival**, **detection**, **qcstats**, **wfdisc**, and **wftag** tables.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Generate and Write TM Beams* [1.3]
- Detailed Description of *Generate Pre-existing Arrival Beams* [1.4]
- Detailed Description of *Perform Detection Processing* [1.5]
- Detailed Description of *Revise Extracted Estimates* [1.6]

- Detailed Description of *Perform Feature Extraction* [1.8]

## General Description

*SeisDet* was designed to detect transient signals in waveforms from a given seismic station and time interval. The station can consist of a single sensor station or an array of sensors. The sensors can be either single-component or three-component (3-C).

## Architecture

*SeisDet* is implemented by a single pass through eight sequential processes (see Figure 12). The first process initializes the site information for this seismic sensor station (1.1). Waveforms for this station and time interval are acquired and checked for quality (1.2). Threshold Monitoring (TM) beams are produced, if required (1.3). Beams for pre-existing arrivals are also generated, if required (1.4). Then detection processing is performed (1.5), and initial values of the primary detection attributes are set in memory (1.6). The next process screens for defective and redundant detections (1.7). Finally, the features are extracted (1.8), and the results are written to the database.

*SeisDet* first accesses the CVAR datastore to extract the station code into the local symbol *sta*. Local symbols, initialized to `nil`, are defined to hold the *initsite* object and the *wfdata* and *det* containers used during subsequent processing. *SeisDet* creates an *initsite* object for the given date by implementing the local Scheme function `create-detection-initsite`. Waveforms are acquired by the function `read-detection-waveforms`.

*SeisDet* then creates and writes TM beams. Detection beams are created for the pre-existing arrivals without beams for the given station and processing time interval. This process is implemented by the local Scheme function `make-pre-existing-arrival-beams`.

## ▼ Seismic Processing

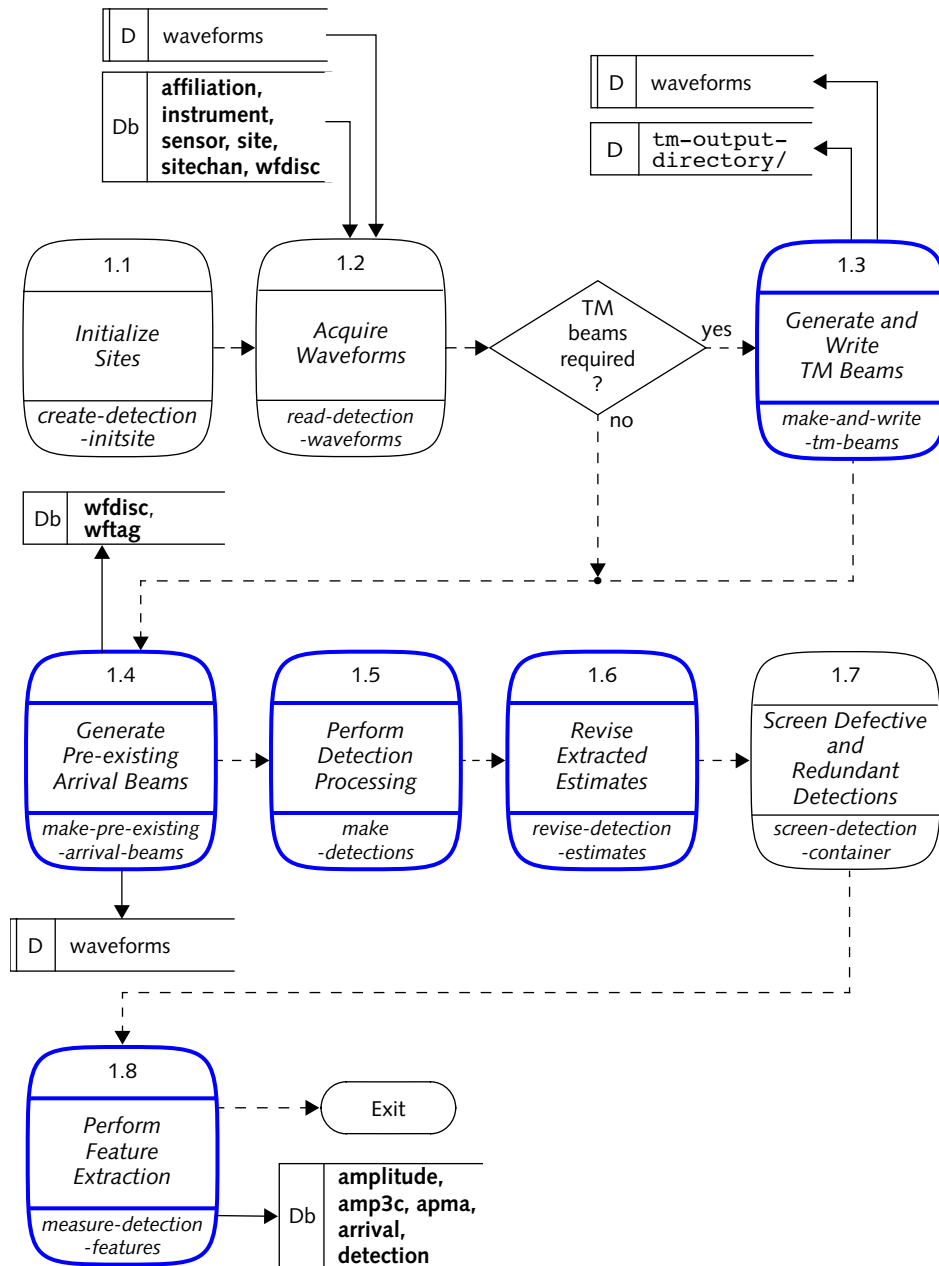


FIGURE 12. ARCHITECTURE OF SEISDET

Next, the actual detections are made, and initial values of some primary detection attributes for each detection are set in memory. These extracted estimates are subsequently revised, if possible, and values for the remaining primary detection attributes are also estimated and initially set in memory. These attributes include: *onset time*, *azimuth*, *slowness*, *amplitude* and *period*. *SeisDet* deletes detections that have parameter values that do not satisfy acceptance criteria, are too close in time to one another, or are too close in time to existing detections.

The next process estimates a variety of secondary features of each detection. The features measured depend on the nature of the seismic sensor(s) at the given station (single sensor, 3-C, sensor array). This process also makes and saves detection beams if sufficient waveforms are available. For detections near the end or beginning of a processing interval sufficient waveforms are sometimes lacking. In these situations, subsequent *DFX* processing for this station and the next processing interval (or the prior processing interval in cases where the pipeline is running backwards) will create the detection beams not already created. For each detection, a pair of matching *dbarrival* and *dbdetection* objects is created, and these objects are written to the appropriate database tables. Amplitude estimates are computed and are likewise written to the appropriate database tables.

## Input/Output

*SeisDet* is called for one station and one time interval at a time. The station code and start-time and end-time parameters are generally specified on the command line. The rest of the parameters are set or referenced in the `DFX-detection.par` file. The input CVAR datastore parameters used in the Scheme code are described in Table 20.

## ▼ Seismic Processing

**TABLE 20: INPUT PARAMETERS FOR AUTOMATIC SEISMIC DETECTION PROCESSING**

Par Name	Source	Default Value	Description
<i>sta</i>	command line	<none>	station code
<i>start-time</i>	command line	<none>	start-time of the processing interval
<i>end-time</i>	command line	<none>	end-time of the processing interval
<i>NetType</i>	par file	<none>	network type, as in: "ss", "array", "larray"
<i>StaType</i>	par file	<none>	station type, as in: "1c", "3c"
<i>WaveType</i>	par file	" "	wave type (" ", "hydro-", "infra-")
<i>wfdisc-extension-len</i>	par file	86400.0	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>data-output-wftag</i>	par file	0	flag for <b>wftag</b> table output: 0 means do not write to <b>wftag</b> table
<i>detection-site-recipe-file</i>	par file	<none>	filename of the site-specific detection recipes
<i>affiliation-table</i>	par file	<b>affiliation</b>	table containing station and network affiliations
<i>instrument-table</i>	par file	<b>instrument</b>	table containing generic (default) calibration information about a station
<i>sensor-table</i>	par file	<b>sensor</b>	table containing specific calibration information for physical channels
<i>site-table</i>	par file	<b>site</b>	table containing site-location information



**TABLE 20: INPUT PARAMETERS FOR AUTOMATIC SEISMIC DETECTION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>sitechan-table</i>	par file	sitechan	table containing station-channel information
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information
<i>out-amplitude-table</i>	par file	amplitude	output table containing amplitude information generated by this application
<i>out-parrival-table</i>	par file	arrival	output table containing predicted arrival information generated by this application
<i>out-detection-table</i>	par file	detection	output table containing signal detection information generated by this application
<i>out-wfdisc-table</i>	par file	wfdisc	output table containing waveform file header and descriptive information generated by this application
<i>out-wftag-table</i>	par file	wftag	output table containing waveform mapping information generated by this application
<i>out-amp3c-table</i>	par file	amp3c	output table containing amplitude estimates (from 3-C stations) generated by this application
<i>out-apma-table</i>	par file	apma	output table containing particle motion analysis information generated by this application
<i>qcstats-table</i> <sup>1</sup>	par file	<none>	table containing statistical measures of the input waveforms developed by the quality-control processing and written by this application

## ▼ Seismic Processing

**TABLE 20: INPUT PARAMETERS FOR AUTOMATIC SEISMIC DETECTION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>qcstats-scheme-file</i> <sup>1</sup>	par file	<none>	filename containing the <i>QCStats</i> source code
<i>perform-qcstats</i>	par file	0 false	flag for generating and writing quality-control statistics: 0 means do not collect data quality statistics
<i>perform-tm-processing</i>	par file	0 false	flag for generating and writing TM beams: 0 means do not perform TM
<i>tm-beam-list</i> <sup>2</sup>	par file	<none>	names of TM beams

1. Optional parameter referenced only if quality-control statistical features are collected.

2. Optional parameter referenced only if TM processing is performed

Table 21 lists the output of *SeisDet*. *SeisDet* writes the detections but first checks the database for detections that already exist by comparing detection times. Those detections that do not already exist are written to the appropriate database tables. *SeisDet* also writes detection beams. There are two output entities per beam: a time-series file (with extension ".w") and a corresponding record to the **wfdisc** table. If the parameter *data-output-wftag* is set to a non-zero value, a corresponding record is written to the **wftag** table for each detection beam.

Table 230 on page 445 lists the specific attributes whose values are written by *SeisDet* for each of these database tables.

**TABLE 21: DATA PRODUCED BY AUTOMATIC SEISMIC DETECTION PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>amplitude</b>	database table	waveform amplitude for each detection
<b>amp3c</b>	database table	waveform amplitude for each detection (on a 3-C station)
<b>apma</b>	database table	particle motion analysis attributes of each detection
<b>arrival</b>	database table	arrival attributes of each detection
<b>detection</b>	database table	detection attributes of each detection
<b>qcstats</b> <sup>1</sup>	database table	statistical attributes of the waveforms developed by the quality-control procedures
<b>wfdisc</b>	database table	detection beams generated
<b>wftag</b> <sup>2</sup>	database table	maps <b>arrival</b> tuples to beams
".w"	binary files	waveforms detection-beam time-series data

1. This product is generated only if the optional *Quality Control Statistics* processing is performed.

2. This product is generated only if the symbol *data-output-wftag* is set non-zero.

The time series for each detection beam is written to the UNIX filesystem in the directory and filenames specified by the *dir* and *dfile* attributes of the **wfdisc** table. The directory name is of the form */base/YYYY/JJJ/* where *JJJ* is the three-digit Julian day of the year (with leading zeros as required), and *YYYY* is the four-digit year for the given input waveforms used to generate the detection beam. The base directory path is set in the *DFX-detection.par* file by the *data-output-base-directory* parameter.

## ▼ Seismic Processing

This application returns an exit status code value that indicates whether or not the processing was successful.

**Processes**

The following processes shown in Figure 12 on page 82 are described in this section:

- *Initialize Sites [1.1]*
- *Acquire Waveforms [1.2]*
- *Generate and Write TM Beams [1.3]*
- *Generate Pre-existing Arrival Beams [1.4]*
- *Perform Detection Processing [1.5]*
- *Revise Extracted Estimates [1.6]*
- *Screen Defective and Redundant Detections [1.7]*
- *Perform Feature Extraction [1.8]*

**Initialize Sites [1.1]**

*SeisDet* requires a variety of parameters for describing the stations to be processed, for acquiring waveforms, and for controlling the processing performed. The *Initialize Sites* process acquires the relevant parameters and binds the Scheme symbols required for subsequent processing. This process is implemented by the local Scheme function `create-detection-initsite`. The function is called by a Scheme language code fragment, such as:

```
(set! initsite-obj (create-detection-initsite sta))
```

The function requires a single argument, which is described in Table 22.

**TABLE 22: FUNCTIONAL INTERFACE OF CREATE-DETECTION-INITSITE**

Argument Name	Usage	Type	Description
<i>sta</i>	input	character-string	station code
n/a	returned value	GObj	<i>initsite</i> object

This function accesses the CVAR datastore to extract the *detection-site-recipe-file*, a variety of database table names, and the *start-time* and *end-time* limits of this processing interval. Table 23 lists the internal symbols that are extracted from the CVAR datastore.

**TABLE 23: INTERNAL SYMBOLS FOR INITIALIZE SITES**

Symbol	Type	Description
<i>detection-site-recipe-file</i>	character-string	filename of the site-specific detection recipes
<i>start-time</i>	real	start-time of this processing interval
<i>end-time</i>	real	end-time of this processing interval
<i>affiliation-table</i>	character-string	table containing station and network affiliations
<i>instrument-table</i>	character-string	table containing generic (default) calibration information about a station
<i>sensor-table</i>	character-string	table containing specific calibration information for physical channels
<i>site-table</i>	character-string	table containing site-location information
<i>sitechan-table</i>	character-string	table containing station-channel information

The local function `create-detection-initsite` follows the general site initialization model described in “Initializing Sites” on page 46. This function creates, sets the attributes in, and returns the *initsite* object based on the recipes from the CVAR datastore for the specified station and time interval. Table 8 on page 50 lists those entities contained in the *initsite* objects created by this function.

If TM processing is required (parameter *perform-tm-processing* is set to unity) then the *tm-rec* is initialized by the Scheme function `init-tm-recipe`. If onset revision is required (parameter *ons-SNR-filter* is set to unity) then both *filterbank-con* and *SNRfilt-rec* are set by the Scheme functions `init-filter-bank` and `init-SNR-filt-recipe`, respectively.

## ▼ Seismic Processing

A key feature of the process *Initialize Sites* is the identification of sensor channels to process. The Scheme function `get-detection-channel-list` creates a list of channels that are used in later processing. This list contains the channel names (for example, "sz" or "be") found in the beam recipes as well as the types of channels required for the processing, specifically applied to various combinations of *nettype* and *statype*. For example, f-k spectra are calculated only for arrays (*nettype* equal to either "array" or "larray"). Therefore, the f-k recipe must be evaluated to ensure that the types of channels used for f-k spectra are acquired if the station is an array. Similarly, 3-C amplitude measurements can be extracted only from 3-C sensors (*statype* equal to 3-C). Consequently, the *amp3c* recipe must be evaluated to ensure that all three channels of a 3-C sensor are acquired if *statype* is equal to 3-C.

This list is used to govern the composition of the *wfdata* container within the *initsite* object. The function is called by the Scheme language code:

```
(set! chan-list (get-detection-channel-list statype
      nettype wavetype beamrec-con fk-rec polar-rec
      amp3creccon))
```

The function requires seven arguments, which are described in Table 24.

**TABLE 24: FUNCTIONAL INTERFACE OF GET-DETECTION-CHANNEL-LIST**

Argument Name	Usage	Type	Description
<i>statype</i>	input	character-string	station type at this station
<i>nettype</i>	input	character-string	network type in which this station is grouped
<i>wavetype</i>	input	character-string	wave type (" ", "hydro-", "infra-")
<i>beamrec-con</i>	input	container	<i>beam recipe</i> objects
<i>fk-rec</i>	input	GObj	<i>f-k recipe</i> object

**TABLE 24: FUNCTIONAL INTERFACE OF GET-DETECTION-CHANNEL-LIST (CONTINUED)**

Argument Name	Usage	Type	Description
<i>polar-rec</i>	input	GObj	<i>polarization processing recipe object</i>
<i>amp3crec-com</i>	input	container	<i>3-C amplitude estimation recipe objects</i>
n/a	returned value	list	character-strings (station codes)

The Scheme function `query-for-wfdata-by-sta-chan-time` returns a *wfdata* container for the stations and channels required to perform the processing. An empty container returned by this function causes *DFX* to exit. Otherwise, the *initsite* attributes are set, and the *initsite* object is returned. At this point in the processing, the *wfdata* container does not yet contain the waveforms.

### Acquire Waveforms [1.2]

*SeisDet* requires waveforms from the stations and channels to be processed to accomplish detection and feature extraction. The process *Acquire Waveforms* obtains the available waveforms in support of subsequent processing. It is implemented by the local Scheme function `read-detection-waveforms`, which is called by the Scheme language code:

```
(read-detection-waveforms initsite)
```

The function requires one argument, which is described in Table 25.

**TABLE 25: FUNCTIONAL INTERFACE OF READ-DETECTION-WAVEFORMS**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
n/a	returned value	container	<i>wfdata</i> objects

## ▼ Seismic Processing

This function follows the general model for acquiring waveforms in *DFX*, which is described in “Reading Waveforms” on page 58.

Determining the appropriate time interval in which to acquire waveforms is a critical facet of *DFX* processing. Two types of factors are used in determining the amount of time by which the nominal processing interval must be extended. The first type are those factors intrinsic to detection processing. These include: (i) the run-up and run-down times for stable spectral filtering, (ii) the run-up times for the short-term and long-term averages, and (iii) the margins required for time re-alignment of waveforms when beam steering. Depending on the processing parameter values and nature of the station geometry, one or another of these may dominate the time extension requirements. The second type are factors controlling sequential processing of consecutive intervals. Processing can be run either in (i) normal sequential order or (ii) reverse sequential order (most recent unprocessed waveforms processed first) when catching up after a downtime in which waveforms accumulated. The nature of the extra waveforms needed to satisfy this requirement is discussed in “Station Processing” on page 65. Both types of factors contribute to altering the time-duration of the interval over which waveforms are acquired for the given processing interval. Specialized library routines are invoked in the course of acquiring waveforms that determine the time-extent of the extra waveforms at each end of the processing interval. The extra waveforms cause the detection processing to be time-overlapped from one processing interval to the next. This may lead to the generation of duplicate detections in successive processing intervals. Consequently, before detection results are written in the database, a search is performed to ensure that the set of detections to be written does not include duplicates of detections already in the database. This is discussed in further detail in “Screen Defective and Redundant Detections [1.7]” on page 98.

The *DFX/libqc* function `qc-waveforms-all` checks the quality of waveforms now contained in the *wfdata* container, which is filled during this process. If parameter *perform-qcstats* is set to a non-zero value, quality-control statistics are produced for this processing station and interval, and this information is written to the *qcstats* database table.



### Generate and Write TM Beams [1.3]

If parameter *perform-tm-processing* is set to a non-zero value, *SeisDet* can generate and save TM beams. TM beams are used to monitor the event detection capability of the primary seismic network. The process *Generate and Write TM Beams* acquires the relevant parameters and binds the Scheme symbols required for subsequent processing.

First, the local Scheme function *instrument-response-files-exist?* is used to determine whether the required instrument response files are accessible for removing instrument response functions from the waveforms. If the required files are not accessible, an error message is printed, and *DFX* exits with a *fatal* status.

The main TM processing is implemented by the local Scheme function *make-and-write-tm-beams*. This process is executed only if the parameter *perform-tm-processing* is set to non-zero. The function is called by the Scheme language code:

```
(make-and-write-tm-beams wfdata-con initsite)
```

The function requires two arguments, which are described in Table 26.

**TABLE 26: FUNCTIONAL INTERFACE OF MAKE-AND-WRITE-TM-BEAMS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>initsite</i>	input	GObj	<i>initsite</i> object
n/a	return value	logical scalar	indicates success (t) or failure (nil)

This function extracts parameters from the CVAR datastore. Then control transfers to a loop over TM beam recipes. For each TM beam recipe, the function performs the following steps:

1. Generate the TM beam from the waveforms in the *wfdata* container in accordance with the current TM beam recipe.

## ▼ Seismic Processing

2. Test whether the generated TM beam is null, and if so, reset the *beam* object to a specially generated null beam.
3. Generate the STA, LTA, and SNR time series for the TM beam.
4. Write the TM beam data to the UNIX filesystem.

If errors occur while generating the STA, LTA, or SNR or while writing the TM beam, an error message is printed, and the loop continues with the next TM beam recipe.

For more information about this function see “Detailed Description of Generate and Write TM Beams [1.3]” on page 101.

**Generate Pre-existing Arrival Beams [1.4]**

For a given processing time interval and station there are a variety of reasons for the prior presence of arrivals in the database. These arrivals generally arise from the extra time-span of actual waveforms processed for each nominal processing time interval (see “Determining Data Time Intervals” on page 64). For any given nominal processing interval there is an overlap of waveforms with both the previous processing interval and the processing interval to follow. These overlaps of waveforms are present to ensure that signals are detected regardless of where they occur within the nominal processing interval. The extra waveforms required for the interval overlap are often insufficient for detection beam generation if the signal arrival is too close to the end of the nominal processing interval. In this case the arrival is written to the database without an associated detection beam and the detection beam is created in the next processing interval.

This process generates and saves detection beams for arrivals that already exist in the database but do not have beams saved for them. A detection beam is a beam that maximizes the coherent signal in the sum of the steered (time-aligned to a given vector slowness) set of waveforms from a group of sensors. The process is implemented by the local Scheme function `make-pre-existing-arrival-beams`, which is called by the Scheme language code:

```
(make-pre-existing-arrival-beams wfdata-con initsite)
```

The function requires two arguments, which are described in Table 27.

**TABLE 27: FUNCTIONAL INTERFACE OF MAKE-PRE-EXISTING-ARRIVAL-BEAMS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>initsite</i>	input	GObj	<i>initsite</i> object
n/a	returned value	list	<i>dbwfdisc</i> and <i>dbwftag</i> objects

This function first ensures that only stations with arrays are processed in it. Then, it extracts parameters from the *initsite* object and queries the database to generate a *dbarrival* container for arrivals that have no detection beams for this station and this processing time interval. For each *dbarrival* object in the container this function performs the following steps:

1. Generate the detection beam from the waveforms in the *wfdata* container in accordance with the information in the beam *ti-recipe* container, the detection recipe, and the arrival time and vector slowness parameter values in the detection object.
2. Write the beam to the appropriate directory and file within the UNIX file-system.
3. Write a corresponding record to the **wfdisc** database table.
4. Test whether the generated beam is null; if so, print an error message. If the parameter *data-output-wftag* is non-zero, then write a corresponding record to the **wftag** database table for each record written to the **wfdisc** database table.

For more information about this function see “Detailed Description of Generate Pre-existing Arrival Beams [1.4]” on page 108.

## ▼ Seismic Processing

**Perform Detection Processing [1.5]**

This process performs detection processing by finding transient signals in the waveforms. It first beamforms the waveforms to a variety of vector slowness values defined in the beam recipes. Then, an STA/LTA detector is applied to each detection beam to find candidate detections (triggers). The triggers are analyzed and, if they satisfy acceptance criteria, detections are generated. This process is implemented by the local Scheme function `make-detections`, which is called by the Scheme language code:

```
(make-detections wfdata-con initsite)
```

The function requires two arguments, which are described in Table 28.

**TABLE 28: FUNCTIONAL INTERFACE OF MAKE-DETECTIONS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>initsite</i>	input and output	GObj	<i>initsite</i> object
n/a	returned value	container	<i>detection</i> objects

This function performs the following steps:

1. Extract parameters from the *initsite* object.
2. In a loop over the beam recipes, generate the beam, and set the results into the symbol *beam*.
3. Generate the STA, LTA, and their quotient SNR for this beam given the detection recipe.
4. If this beam is a standard beam, insert it into the standard-beam container (which is returned as a side-effect because it is referenced as a field of the *initsite* argument).
5. If this beam is a detecting beam then evaluate the beam for triggers by the *DFX/libdet* function `find-triggers`, and return these triggers in a *trigger* container.

6. If the *trigger* container is not null, (*DFX/libdet* function *find-detections*) evaluate the *trigger* container. If the triggers satisfy acceptance criteria for declaration of a detection, return a detection container. See [IDC5.2.1] for more information about seismic STA/LTA detectors.

For more information about this function see “Detailed Description of Perform Detection Processing [1.5]” on page 113.

### Revise Extracted Estimates [1.6]

*SeisDet* performs detection processing by finding, and measuring both the primary and secondary attributes of, transient signals in the waveforms. In many instances, some of the primary attributes initially measured are only roughly estimated. Consequently, refined estimates are required. In addition, several of the primary detection attributes are not initially measured in the process of declaring detection. The process *Revise Extracted Estimates* refines the initially rough estimates and extracts estimates for those primary detection attributes that have not been set. This is accomplished via the local Scheme function *revise-detection-estimates*. This function is called by the Scheme language code:

```
(revise-detection-estimates wfdata-con initsite det-con)
```

This function requires three arguments, which are described in Table 29.

**TABLE 29: FUNCTIONAL INTERFACE OF REVISE-DETECTION-ESTIMATES**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>det-con</i>	input and output	container	<i>detection</i> objects
n/a	returned value	container	<i>detection</i> objects

## ▼ Seismic Processing

For the given station and processing time interval this function extracts parameters from the *initsite* object and creates a defective-detection container (symbol *bad-det-con*). This container is created to hold detection objects that are to be eliminated from the set of input detection objects. In a loop over detection objects this function performs the following steps:

1. Revise the detection onset time estimate.
2. Estimate the detection amplitude.
3. In the case of array stations, generate an f-k spectrum.
4. In the case of stations with 3-C sensors, perform polarization analysis.

Within this processing loop, if any failures in measurement result, the detection is deleted, and a warning message about the nature of the failure and the identity of the deleted detection is printed. This function returns the input detection container with deleted detections removed from it.

For more information about this function see “Detailed Description of Revise Extracted Estimates [1.6]” on page 122.

### Screen Defective and Redundant Detections [1.7]

*SeisDet* produces detections only where the waveforms support good measurements. Detections for which important parameters cannot be successfully measured are deemed defective and are discarded. Furthermore, to guarantee that there are no lost detections at the edges of successive processing intervals, the successive processing intervals are overlapped in time. Consequently, a signal may be detected in two successive processing intervals if it occurs in the time-overlap portions of each processing interval. To prevent redundant detection records in the database, the detections from each processing interval are screened to eliminate those that already exist in the database; that is, newly generated redundant detections are discarded. The process *Screen Defective and Redundant Detections* accomplishes this. It is implemented by the local Scheme function `screen-detection-container`. This function is called by the Scheme language code:

```
(screen-detection-container initsite det-con)
```

The function requires two arguments, which are described in Table 30.

**TABLE 30: FUNCTIONAL INTERFACE OF SCREEN-DETECTION-CONTAINER**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>det-con</i>	input and output	container	<i>detection</i> objects
n/a	returned value	container	<i>detection</i> objects

This function performs the following steps:

1. Screen the detections to eliminate those that fail to meet acceptance criteria or are too closely spaced to one another in time.
2. Screen the detections to eliminate those that match detections already in the database. For more information see “Internal Screening of Containers” on page 74. This function returns the input detection container with screened detections deleted from it.

### Perform Feature Extraction [1.8]

*SeisDet* produces measurements of signal/background features for each detection it produces. The process *Perform Feature Extraction* estimates the waveform and background attributes in the vicinity of the detections. The process is implemented by the local Scheme function `measure-detection-features`. This function is called by the Scheme language code:

```
(measure-detection-features wfddata-con initsite det-con)
```

The function requires three arguments, which are described in Table 31.

## ▼ Seismic Processing

TABLE 31: FUNCTIONAL INTERFACE OF MEASURE-DETECTION-FEATURES

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>det-con</i>	input	container	<i>detection</i> objects
n/a	returned value	container	<i>detection</i> objects

This function makes decisions based upon the network type and station type as to what sort of features can be extracted from the waveforms. First it extracts parameters both from the CVAR datastore and from the *initsite* object. Then it loops over the *detection* objects contained in the input *detection* container. Within this loop the process performs the following steps:

1. Generate a *dbarrival* object from the *detection* object, and write this *dbarrival* object to the **arrival** database table.
2. If the station is a 3-C seismic sensor, generate a *dbapma* object from the *detection* object, and write this *dbapma* object to the **apma** database table.
3. Estimate detection amplitudes in accordance with the amplitude recipes, and write this *dbamplitude* container to the **amplitude** database table.
4. Generate detection beams in accordance with the detection recipe and beam ti-recipes.
5. Generate a *dbdetection* object from the *detection* object, and write this *dbdetection* object to the **detection** database table.
6. Generate *sbsnr dbamplitude* objects from the *detection* object, and write the *sbsnr dbamplitude* container to the **amplitude** database table.
7. If the station consists of a single 3-C seismic sensor, estimate 3-C amplitudes, and write the *dbamplitude* container to the **amp3c** database table.



In any case where an error is encountered, either in generating information or in submitting this information to the database, the function prints an error message, further processing stops, the database is rolled-back, and control passes to the top of the loop over detections. When all feature extraction processes are successful, the database is committed, and control is returned to the calling function.

For further information about this function see “Detailed Description of Perform Feature Extraction [1.8]” on page 132.

### **Detailed Description of Generate and Write TM Beams [1.3]**

This process generates and writes TM beams by making a single pass through a single initialization subprocess followed by a loop over the TM recipes wherein the TM beams are generated and saved. Figure 13 shows the architecture of the process. The subprocess *Extract Parameters and Save and Reset stav-len* extracts parameters and resets the detection recipe parameter *det-stav-len* for TM beam processing. Within a loop over TM beam recipes the following subprocesses are invoked for each TM beam recipe: *Generate Beam*, *Generate STA and LTA* (and their quotient, SNR), and *Write TM Beam* (to UNIX filesystem). When all suitable beam recipes have been used and the loop is complete, the process resets the *det-stav-len* in the CVAR datastore to the original automatic seismic data processing value it held prior to being set to a TM value, and control is returned to the calling function.

The following subprocesses shown in Figure 13 are described in this section:

- *Extract Parameters and Save and Reset stav-len [1.3.1]*
- *Generate Beam [1.3.2]*
- *Generate Null Beam [1.3.3]*
- *Generate STA and LTA [1.3.4]*
- *Write TM Beam to UNIX Filesystem [1.3.5]*

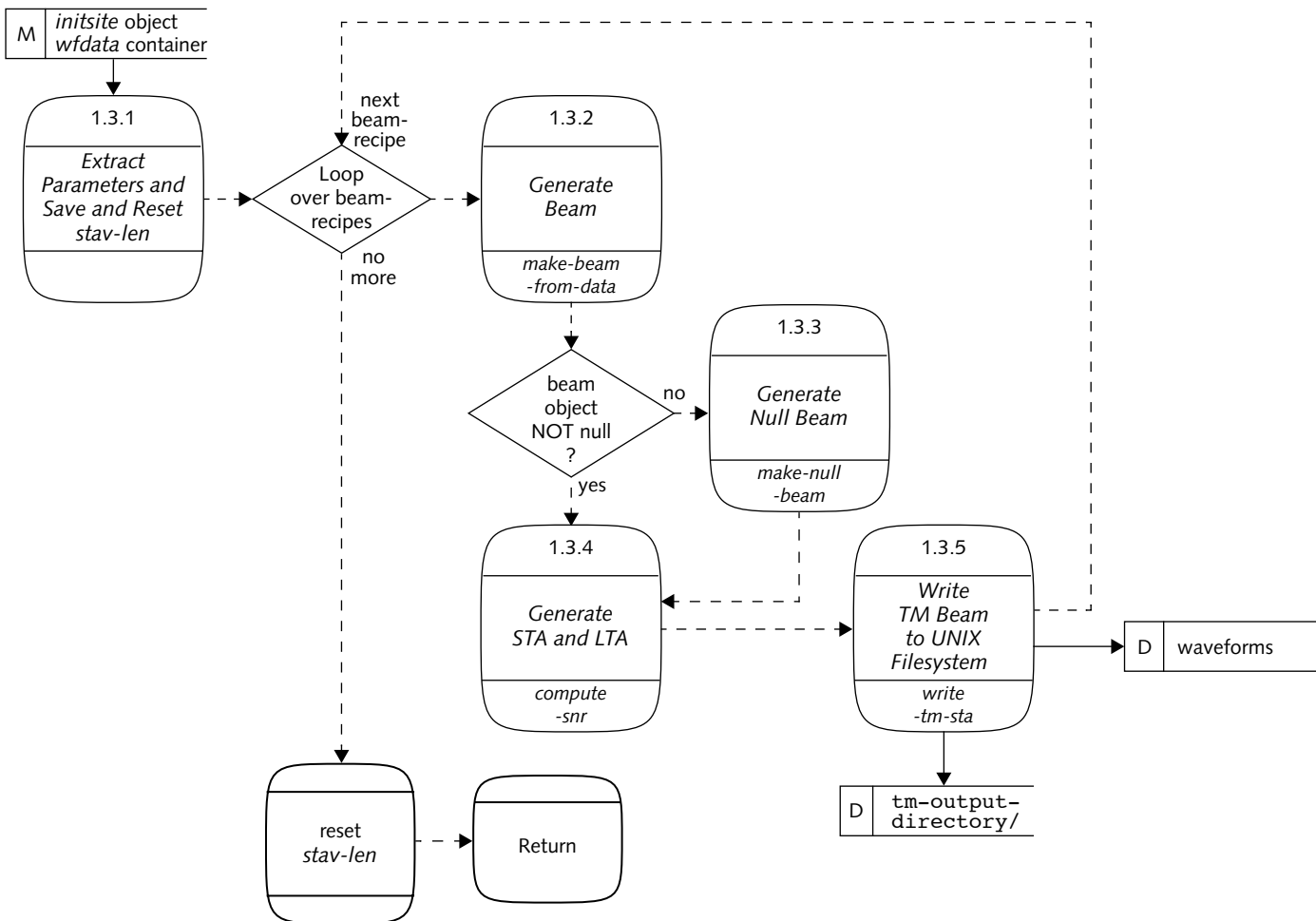


FIGURE 13. ARCHITECTURE OF GENERATE AND WRITE TM BEAMS

### Extract Parameters and Save and Reset stav-len [1.3.1]

The process *Generate and Write TM Beams* requires a variety of parameters and recipes for its activities. The subprocess *Extract Parameters and Save and Reset stav-len* acquires the parameters and recipes required to generate and write the TM beams. The subprocess is implemented by the intrinsic Scheme function `let*`, the global Scheme functions `getspar`, `delimited-string->list`, and `scan-container`, and the common *libgobj* Scheme function `extract-gobj-attr`. It accesses the parameters in the CVAR datastore and the *initsite* object, and creates the local symbols listed in Table 32.

**TABLE 32: PARAMETERS ACQUIRED BY EXTRACT PARAMETERS AND SAVE AND RESET STAV-LEN**

Parameter Name	Source
<i>tm-beam-list</i>	CVAR
<i>beamrec-con</i>	<i>initsite</i> object
<i>tm-beamrec-con</i>	CVAR and <i>initsite</i> object
<i>det-rec</i>	<i>initsite</i> object
<i>tm-rec</i>	<i>initsite</i> object
<i>start-time</i>	<i>initsite</i> object
<i>end-time</i>	<i>initsite</i> object
<i>tm-stav-len</i>	<i>initsite</i> object
<i>outdir</i>	<i>initsite</i> object
<i>step</i>	<i>initsite</i> object
<i>srate</i>	<i>initsite</i> object
<i>nullval</i>	<i>initsite</i> object
<i>cflength</i>	<i>initsite</i> object
<i>save-stavlen</i>	<i>initsite</i> object

## ▼ Seismic Processing

**Generate Beam [1.3.2]**

The process *Generate and Write TM Beams* generates TM beams formed in accordance with appropriate beam recipes from suitably selected time intervals of waveforms. The subprocess *Generate Beam* generates each TM beam. It is implemented by the global Scheme function `make-beam-from-data`. This function is called by the Scheme language code:

```
(make-beam-from-data wfdata-con beamrec)
```

The function requires two arguments, which are described in Table 33.

**TABLE 33: FUNCTIONAL INTERFACE OF MAKE-BEAM-FROM-DATA**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>beamrec</i>	input	GObj	<i>beam recipe</i> object
n/a	returned value	GObj	<i>beam</i> object

The function generates a beam from the available waveforms in accordance with the beam recipe using the *DFX/libbeam* function `make-beam` to actually generate the beam time series. The function `make-beam-from-data` sets up the arguments required for `make-beam` via direct access to the CVAR datastore as well as the contents of the *wfdata* container. This function returns a *beam* object.

**Generate Null Beam [1.3.3]**

The process *Generate and Write TM Beams* generates TM beams formed in accordance with appropriate beam recipes from suitably selected time intervals of waveforms. In some instances a null beam is returned by the subprocess *Generate Beam*. When this occurs, subprocess *Generate Null Beam* is called by the *DFX/libbeam* function `make-null-beam` to generate a *beam* object of valid structure. This function is called by the Scheme language code:

```
(make-null-beam beamrec start-time end-time srate)
```

The function requires three arguments, which are described in Table 34.

The function generates and returns a *beam* object that has all of the time-series data arrays for the beam, the STA, the LTA, the SNR, and so on, with allocated memory sufficient for the time-duration of the beam that was intended to be generated in the subprocess *Generate Beam*. Instead of useful waveform-derived values in these time series, the time series have floating-valued zeros written in all of their elements.

**TABLE 34: FUNCTIONAL INTERFACE OF MAKE-NULL-BEAM**

Argument Name	Usage	Type	Description
<i>beamrec</i>	input	GObj	<i>beam recipe</i> object
<i>start-time</i>	input	real	start-time of the null beam to be created
<i>end-time</i>	input	real	end-time of the null beam to be created
<i>srate</i>	input	real	sample rate of the null beam to be created
n/a	returned value	GObj	<i>beam</i> object

### Generate STA and LTA [1.3.4]

The process *Generate and Write TM Beams* generates TM beams formed in accordance with appropriate beam recipes from suitably selected time intervals of waveforms. In the context of *DFX*, a *beam* object consists of several data arrays in parallel: the beam time series itself, a time series of STA beam magnitudes (L1 norm) or power levels (L2 norm), a time series of matching LTA beam values, a time series of their ratio,  $SNR = STA/LTA$ , and so on. The subprocess *Generate STA and LTA* generates these additional time series from the original beam time series. It is implemented by the *DFX/libdetect* function `compute-snr`. This function is called by the Scheme language code:

```
(compute-snr beam det-rec)
```

## ▼ Seismic Processing

The function requires two arguments, which are described in Table 35.

**TABLE 35: FUNCTIONAL INTERFACE OF COMPUTE-SNR**

Argument Name	Usage	Type	Description
<i>beam</i>	input and output	GObj	<i>beam</i> object
<i>det-rec</i>	input	GObj	<i>seismic detection processing recipe</i> object
n/a	returned value	GObj	<i>beam</i> object

The function creates the principal output for the process *Generate and Write TM Beams*. It creates the STA, LTA, SNR, and state time series from the given beam time series in accordance with the parameters in the detection recipe. The returned value from `compute-snr` is the input *beam* object enhanced by having several of its attributes reference data arrays (of time series with memory allocated) generated by this function: *norm*, *stav*, *ltav*, *snr*, and *state*. Although the STA, LTA, SNR, and state time series are generated, only the STA time series is saved as the beam by the TM processing.

### Write TM Beam to UNIX Filesystem [1.3.5]

The process *Generate and Write TM Beams* generates and saves TM beams. This subprocess saves the generated TM beams in the UNIX filesystem in the appropriate directory in a circular file. It is implemented by the *DFX/lbio* function `write-tm-sta`. This function is called by the Scheme language code:

```
(write-tm-sta beam outdir start-time end-time step nullval
  cflength)
```

The function requires seven arguments, which are described in Table 36.

TABLE 36: FUNCTIONAL INTERFACE OF WRITE-TM-STA

Argument Name	Usage	Type	Description
<i>beam</i>	input	GObj	<i>beam</i> object
<i>outdir</i>	input	character-string	directory name where TM beams are to be written
<i>start-time</i>	input	real	start-time of this processing interval
<i>end-time</i>	input	real	end-time of the processing interval
<i>step</i>	input	real	inter-sample time increment (1/sample rate)
<i>nullval</i>	input	real	value to use in the STA array if there are no data (in case of a null beam)
<i>cflength</i>	input	real	number of samples to write to each TM beam file
n/a	returned value	real	double precision floating value $\geq 0$ or <code>nil</code> (indicates error)

This function writes each TM beam, which is actually a beam STA time series, to the UNIX filesystem in the directory indicated by the symbol *outdir* with a suitable filename extracted from the beam recipe. The function `write-tm-sta` returns a value of `nil` if an error occurred; otherwise, it returns a double-precision floating number of value zero or greater.

### Detailed Description of Generate Pre-existing Arrival Beams [1.4]

This process generates and writes beams for pre-existing arrivals within the processing time interval in every case where the arrival has no corresponding detection beam. The process is implemented by the local Scheme function `make-pre-existing-arrival-beams`.

Figure 14 shows the architecture of *Generate Pre-existing Arrival Beams*. The process is implemented by a single pass through the initialization subprocesses *Extract Parameters* and *Query Database* (arrivals with beams), followed by a loop over pre-existing arrivals. Within the loop, the following subprocesses are performed for each pre-existing arrival: *Generate Detection Object*, *Extract Parameters from Dbarrival Object (and Set into Detection Object)*, and finally, *Generate Detection Beam*. When all arrivals have been processed control is returned to the calling function.

The following subprocesses shown in Figure 14 are described in this section:

- *Extract Parameters* [1.4.1]
- *Query Database (Arrivals with Beams)* [1.4.2]
- *Generate Detection Object* [1.4.3]
- *Extract Parameters from Dbarrival Object (and Set into Detection Object)* [1.4.4]
- *Generate Detection Beam* [1.4.5]

#### Extract Parameters [1.4.1]

To generate beams for pre-existing arrivals that do not already have associated beams saved, *DFX* requires a variety of parameter values and symbols to be set. The subprocess *Extract Parameters* acquires and binds these parameter values and symbols. It is implemented by the intrinsic and common *libgobj* Scheme function `let*` and `extract-gobj-att`, respectively. The subprocess accesses the parameters in the *initsite* object and creates the local symbols listed in Table 37.



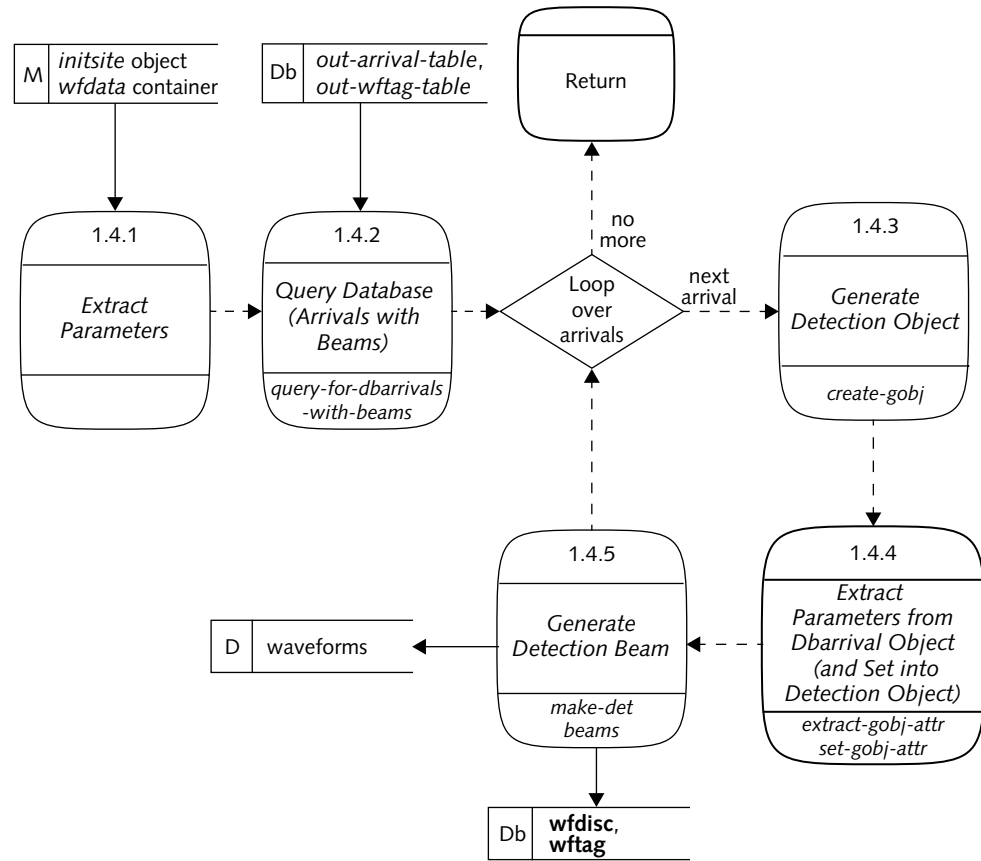


FIGURE 14. ARCHITECTURE OF GENERATE PRE-EXISTING ARRIVAL BEAMS

TABLE 37: PARAMETERS ACQUIRED BY EXTRACT PARAMETERS

Parameter Name	Source
<i>nettype-ss?</i>	<i>init site object</i>
<i>sta</i>	<i>init site object</i>
<i>beamtirec-con</i>	<i>init site object</i>
<i>det-rec</i>	<i>init site object</i>

## ▼ Seismic Processing

This subprocess is executed only when the station being processed is an array station, as indicated in the source code when the symbol *nettype-ss?* is valued *nil* (the station is a single sensor).

### Query Database (Arrivals with Beams) [1.4.2]

To generate pre-existing arrival beams for arrivals that do not already have saved beams, *DFX* must identify all of the arrivals for this time interval and station and must also identify all of the arrivals for this time interval and station that have saved beams. The subprocess *Query Database (Arrivals with Beams)* accomplishes these steps. It is implemented by the local Scheme function *query-for-dbarrivals-with-beams*, which is called by the Scheme language code:

```
(query-for-dbarrivals-with-beams wfdata-con initsite)
```

The function requires two arguments, which are described in Table 38.

**TABLE 38: FUNCTIONAL INTERFACE OF QUERY-FOR-DBARRIVALS-WITH-BEAMS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input and output	container	<i>wfdata</i> objects
<i>initsite</i>	input	GObj	<i>initsite</i> object
n/a	returned value	container	<i>dbarrival</i> objects

This function returns a *dbarrival* container from the **out-arrival-table** and **out-wftag-table**. The returned *dbarrival* objects are constrained to be in the current processing time interval (adjusted for lead and lag) for the current station. The **out-wftag-table** is queried to find those arrivals for which there are no beams. Thus, the *dbarrival* objects returned are those for which beams need to be generated. For general information on how database queries are implemented see “Querying the Database” on page 41.

**Generate Detection Object [1.4.3]**

The main activity of the process *Generate Pre-existing Arrival Beams* is to generate detection beams. The functions that generate detection beams require a *detection* object for each detection beam generated. Thus, to use these functions each arrival must have a corresponding *detection* object. The subprocess *Generate Detection Object* generates a *detection* object for each arrival to be processed. It is implemented by the common *libgobj* Scheme function `create-gobj`. In addition to creating the local *detection* object (symbol *det*), this function creates an initially nil-valued symbol *wflist* that is later used to hold the generated beams.

**Extract Parameters from Dbarrival Object (and Set into Detection Object) [1.4.4]**

The main activity of the process *Generate Pre-existing Arrival Beams* is to generate detection beams. The function for beam generation requires, as one of its input arguments, a *detection* object. Therefore, a *detection* object must be generated from the available information, which can be found in the *dbarrival* object. The subprocess *Generate Detection Object* creates the required *detection* object. The subprocess *Extract Parameters from Dbarrival Object (and Set into Detection Object)* fills the essential attributes in the *detection* object with parameter values it acquires from the corresponding *dbarrival* object. It is implemented by the inline generic Scheme functions `let`, `extract-gob-attr`, and `set-gobj-attr`. The attributes of the *detection* object set from the contents in the *dbarrival* object are listed in Table 39.

**TABLE 39: PARAMETERS SET IN LOCAL DETECTION GOBJ**

Detection Parameters Set	Dbarrival Parameters Accessed
<i>arid</i>	<i>arid</i>
<i>time</i>	<i>time</i>
<i>seaz</i>	<i>azimuth</i>
<i>slow</i> (sec/km)	<i>slow</i> (sec/degree)

## ▼ Seismic Processing

**Generate Detection Beam [1.4.5]**

The main activity of the process *Generate Pre-existing Arrival Beams* is to generate detection beams. The subprocess *Generate Detection Beam* produces the detection beam from the waveforms. It is implemented by the local Scheme function `make-det-beams`. This function is called by the Scheme language code:

```
(make-det-beams wfdata-con beamtirec-con det-rec det)
```

This function requires four arguments, which are described in Table 40.

**TABLE 40: FUNCTIONAL INTERFACE OF MAKE-DET-BEAMS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>beamtirec-con</i>	input	container	<i>beam time-interval recipe</i> objects
<i>det-rec</i>	input	GObj	<i>seismic detection processing recipe</i> object
<i>det</i>	input	GObj	<i>detection</i> object
n/a	returned value	list	<i>beam</i> objects

This function returns a list of the *detection beam* objects. It performs the following steps:

1. Extract parameters from the input *detection* object.
2. Test whether or not the slowness required in the input *detection* object satisfies the criterion *det-output-beam-max-slow* from the input detection recipe.
3. Where the slowness criterion is satisfied, create a local *dbwfdisc* container and a local *dbwftag* container.
4. Loop over the input beam time-interval recipe container. For each *beam time-interval recipe* object, obtain an unique waveform identification number from Scheme function `query-for-last-id` for the beam about to be created.

5. Set a local beam recipe container to the detection beam recipes listed in the time-interval recipes via the global Scheme function `get-beamrec-for-tirec-from-beamtirec-con`.
6. Create a detection beam *wfdisc* container via the *DFX/libio* function `create-db-detection-beam`.
7. Insert this beam container into the local *dbwfdisc* container via the global Scheme function `insert-container-sub`.
8. When the loop over all available *beam time-interval recipe* objects is completed, submit the collection of detection beam *dbwfdisc* and *dbwftag* objects to the database via the local Scheme function `submit-dbwf-disc-dbwftag`. In this last step, the function submits both the *dbwfdisc* container and the *dbwftag* container to the appropriate database tables specified by the parameters *out-wfdisc-table* and *out-wftag-table*. For more information on submitting results to the database see “Writing to the Database” on page 44.

### Detailed Description of Perform Detection Processing [1.5]

This process identifies detections within the processing time interval, the first step in creating the final detection set. It is implemented by a call to the local Scheme function `make-detections`.

Figure 15 shows the architecture of *Perform Detection Processing*. The process is implemented by a single pass through the initialization subprocess *Extract Parameters*, followed by a loop over standard beam and detection beam recipes. Within the loop, for each detection beam recipe, the process extracts the beam name, generates the detection beam, generates the STA and LTA (and SNR and state) time series for the beam, and generates triggers. When all detection beam recipes have been processed and the loop is completed, the subprocess evaluates the triggers and generates the detections. A *detection* container is returned to the calling function.

## ▼ Seismic Processing

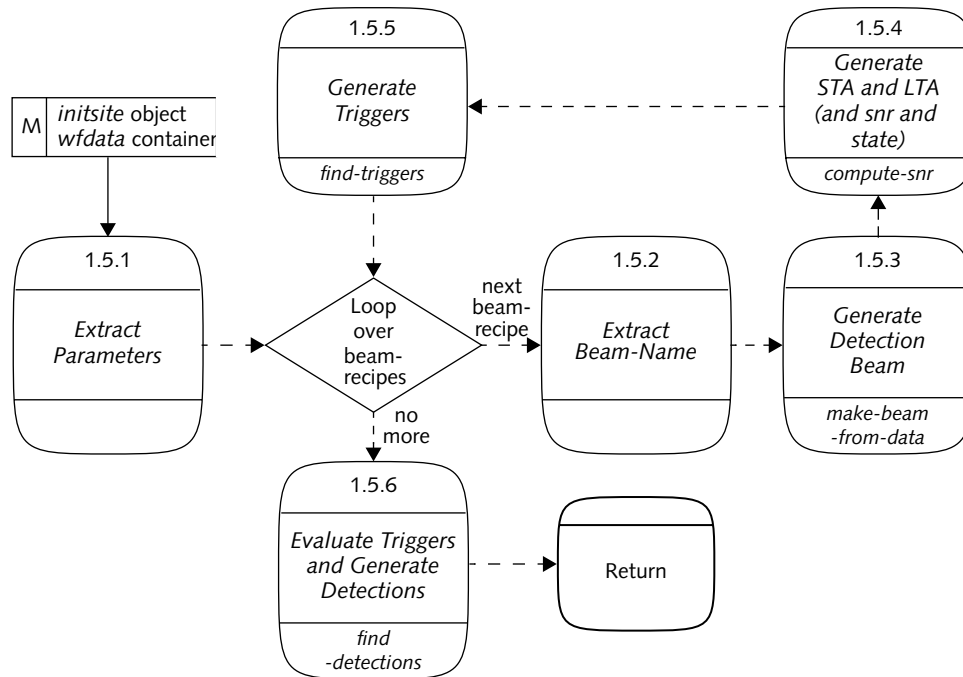


FIGURE 15. ARCHITECTURE OF PERFORM DETECTION PROCESSING

The following subprocesses shown in Figure 15 are described in this section:

- *Extract Parameters [1.5.1]*
- *Extract Beam-Name [1.5.2]*
- *Generate Detection Beam [1.5.3]*
- *Generate STA and LTA (and SNR and state) [1.5.4]*
- *Generate Triggers [1.5.5]*
- *Evaluate Triggers and Generate Detections [1.5.6]*

**Extract Parameters [1.5.1]**

The process *Perform Detection Processing* requires a variety of parameter values and symbols to be set for it to operate. The subprocess *Extract Parameters* acquires and binds these parameter values and symbols. It is implemented by the intrinsic

Scheme function `let` and common *libgobj* `extract-gobj-att`, respectively. It accesses the attributes in the *initsite* object and creates the local symbols listed in Table 41. This function also creates an initially empty *trigger* container (symbol *trig-con*).

**TABLE 41: PARAMETERS ACQUIRED BY EXTRACT PARAMETERS**

Parameter Name	Source
<i>stdbeam-con</i>	<i>initsite</i> object
<i>beamrec-con</i>	<i>initsite</i> object
<i>det-rec</i>	<i>initsite</i> object
<i>start-time</i>	<i>initsite</i> object
<i>end-time</i>	<i>initsite</i> object

### **Extract Beam-Name [1.5.2]**

The process *Perform Detection Processing* generates detection beams and evaluates these beams to find detections. The detection beam must be named. The subprocess *Extract Beam-Name* acquires a suitable detection beam name from the *beam-name* attribute of the appropriate *beam recipe* object. It also creates an error message for use in case subsequent functions fail. This subprocess is implemented by the intrinsic and common *libgobj* Scheme functions `let*` and `extract-gobj-attr`, respectively.

Following this function, an initially nil local symbol *beam* is created, and an explicit Scheme environment “garbage collection” is performed by the intrinsic Scheme function `gc`.

## ▼ Seismic Processing

**Generate Detection Beam [1.5.3]**

The process *Perform Detection Processing* generates beams and evaluates the detection beams among them to find detections. The subprocess *Generate Detection Beam* creates a beam from the waveforms in accordance with the current *beam recipe* object. It is implemented by the global Scheme function `make-beam-from-data`. This function is called by the Scheme language code:

```
(make-beam-from-data wfdata-con beamrec)
```

The function requires two arguments, which are described in Table 42.

**TABLE 42: FUNCTIONAL INTERFACE OF MAKE-BEAM-FROM-DATA**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>beamrec</i>	input	GObj	<i>beam recipe</i> object
n/a	returned value	GObj	<i>beam</i> object

The function `make-beam-from-data` provides a “wrapper” to the *DFX/libbeam* function `make-beam`. The function `make-beam` accesses and processes the waveforms to form the beam. It returns a *beam* object. If the *beam recipe* object defines a standard beam this beam is saved in a standard beams container and is returned from the process *Perform Detection Processing* as a side-effect via the *initsite* object. If the beam recipe requires a detection beam, then this beam is subject to further evaluation for generating detections.

**Generate STA and LTA (and SNR and state) [1.5.4]**

The process *Perform Detection Processing* generates and evaluates detection beams to find detections. The evaluation is based upon a time series of detection statistic values generated from the time series of each detection beam. The detection statistic currently in use is an L1-norm STA of beam time-series sample magnitudes compared with the LTA of beam time-series sample magnitudes. The magnitudes



are compared by forming the time series of the sample-by-sample ratio of the time series of STA to the time series of LTA. This ratio time series is the SNR. In the presence of ambient noise alone, free of seismic events, the SNR should be approximately unity. When a transient seismic signal arrives the ratio should become significantly larger than unity.

The time series of beamformed data that are used to generate the STA and LTA time series are computed from a sum over time-aligned waveform samples (or a transformation of waveforms) from several sensors of a sensor array. In general, each sensor's waveforms have a pattern of defects that may be independent of the other sensors. Because waveform samples that have been deemed defective (via the quality-control processing) are not summed into the beam time series, the number of sensors contributing to the sum may vary. The "raw" beam generated in this fashion is not useful unless a correction is made for the number of sensors contributing to each time-sample. This correction is accomplished by a time series in the *beam* object called the "norm" array. It is a time series of the count of sensors corresponding to the beamformed sum time series. For samples where the norm is less than a threshold value, the given sample is ignored in subsequent processing such as in the calculation of the STA and LTA time series. The threshold for using a beam sample is the product of the detection recipe parameter *det-min-sta-fraction* and the number of elements in the beam sum (sensors if 1-C stations or, in the case of 3-C stations, channels adjusted for beam rotation, if any). In no case is the product allowed to drop below unity. To facilitate application of the norm time-series and the threshold applied to it, another time-series array, the state, is created and stored in parallel with the beam. The state array is unity valued for all time samples where the beam's norm is above threshold, and it is zero-valued elsewhere. The state array is analogous to that of the mask time-series array for an individual channel of waveforms. Thus, in the actual generation of the STA and LTA time series, the value of the state array is used in conjunction with each corresponding beam datum. This ensures that the STA, LTA, and SNR are not used to generate faulty triggers and ultimately unreliable detections.

The subprocess *Generate STA and LTA (and SNR and state)* creates a time series of STA values, LTA values, SNR values, and state values. These four time series are used for detection processing. The STA, LTA, SNR, and state are formulated in

## ▼ Seismic Processing

accordance with the parameter value specified in the detection recipe. This subprocess is implemented by the *DFX/libdetect* function `compute-snr`. This function is called by the Scheme language call:

```
(compute-snr beam det-rec)
```

The function requires two arguments, which are described in Table 43.

**TABLE 43: FUNCTIONAL INTERFACE OF COMPUTE-SNR**

Argument Name	Usage	Type	Description
<i>beam</i>	input and output	GObj	<i>beam</i> object
<i>det-rec</i>	input	GObj	<i>seismic detection processing recipe</i> object
n/a	returned value	GObj	<i>beam</i> object

The returned value from `compute-snr` is the input *beam* object enhanced by having several of its attributes set: *stav*, *ltav*, *snr*, and *state*. If the input *beam* object is a *detection-beam* object, then the enhanced version of it is subject to further processing to develop detections. If the input *beam* object is a *standard-beam* object, then the enhanced version of it is inserted into the *standard-beam* container. Because the *standard-beam* container is an attribute in the *initsite* object, its information is returned to the calling program as a side-effect.

### Generate Triggers [1.5.5]

The process *Perform Detection Processing* generates and evaluates detection beams to find detections. The evaluation is based upon a time series of detection statistic values (the SNR). The subprocess *Generate Triggers* evaluates the SNR time series in comparison to a fixed threshold (from the *beam recipe* object used to specify the detection beam). The SNR time series is evaluated in a loop running over its samples. Wherever the SNR value exceeds the threshold and also satisfies other detec-

tion criteria, this subprocess generates a *trigger* object. The *trigger* objects are saved in a *trigger* container. This function is implemented by the *DFX/libdetect* function `find-triggers`. This function is called by the Scheme language code:

```
(find-triggers beam start-time end-time trig-con det-rec)
```

The function requires five arguments, which are described in Table 44.

**TABLE 44: FUNCTIONAL INTERFACE OF FIND-TRIGGERS**

Argument Name	Usage	Type	Description
<i>beam</i>	input	GObj	<i>beam</i> object
<i>start-time</i>	input	real	start-time for trigger processing
<i>end-time</i>	input	real	end-time for trigger processing
<i>trig-con</i>	output	container	<i>trigger</i> objects
<i>det-rec</i>	input	GObj	<i>seismic detection processing recipe</i> object
n/a	returned value	container	<i>trigger</i> objects, <i>nil</i> (indicates error)

The returned value from `find-triggers` is the input *trigger* container or *nil* if an error is encountered.

### Evaluate Triggers and Generate Detections [1.5.6]

The process *Perform Detection Processing* generates and evaluates detection beams to find detections. The subprocess *Evaluate Triggers and Output Detections* finds detections by evaluating the triggers previously generated. This subprocess is implemented by the *DFX/libdetect* function `find-detections`. This function is called by the Scheme language code:

```
(find-detections trig-con det-rec)
```

## ▼ Seismic Processing

The function requires two arguments, which are described in Table 45.

**TABLE 45: FUNCTIONAL INTERFACE OF FIND-DETECTIONS**

Argument Name	Usage	Type	Description
<i>trig-con</i>	input	container	<i>trigger</i> objects
<i>det-rec</i>	input	GObj	<i>seismic detection processing recipe</i> object
n/a	returned value	container	<i>detection</i> objects nil (indicates error)

The function `find-detections` evaluates each of the *trigger* objects in the *trigger* container in accordance with the detection algorithm and the parameter values found in the detection recipe. Where possible it generates a *detection* object and saves the object in a *detection* container. The returned value from `find-detections` is a *detection* container, which is empty if no trigger can satisfy the criteria for detection or `nil` if an error is encountered.

The following algorithm is used to declare detections:

1. Sort the triggers in the input *trigger* container by time from earliest to latest.
2. Determine the number of samples from the onset of the first trigger to the shut-off of the last trigger. Allocate a detection state vector of this length.
3. Compute the detection state vector as a sum over all the triggers.
4. Determine the interval length for declaring new detections.
5. Find detection triggers. Evaluate each point in the detection state vector. A detecting state occurs when state value exceeds *min\_triggered\_beams*. Save the first trigger beam recipe. If the state value is in a state of detection and the time since the start of this detecting interval is less than *min\_det\_interval*, increment the interval and continue. If the state value has fallen below *min\_triggered\_beams*, a detecting interval is completed. Turn off detecting state, and look for the best SNR trigger within this interval. Track the current trigger for the next time. Add the best SNR trigger to the dynamic array. If a detecting interval is not completed, continue. If it is not in a detecting state, check if the next sample is in a detection state.
6. If all samples of the state vector have been evaluated and the prevailing state is "detecting," complete the last *detection* object.
7. Set the array of detection structures for output.

See [IDC5.2.1] for more information on making seismic detections.

### Detailed Description of Revise Extracted Estimates [1.6]

This process provides initial estimates of the onset time, amplitude(s), period, azimuth and slowness, and where applicable, polarization for those detections present at the given station for the given processing time interval. Where any of these parameters cannot be estimated, the detection is deleted. Thus, on completion, this process returns a revised *detection* container, that is, *detection* objects with feature measurements.

Figure 16 shows the architecture of *Revise Extracted Estimates*. The process is implemented by a single pass through the initialization subprocess *Extract Parameters*, followed by a loop over *detection* objects, and concluded by removing defective *detection* objects. Figure 16 uses a special symbology in which control passes from many points to a process labeled “K” where the current detection object is inserted into a detection container that holds those detection objects that are defective or incomplete. Once the “bad” detection object is inserted in this container, control returns to the top of the loop over detections. For each *detection* object within the main loop over *detection* objects the process performs the following steps:

1. Revise the onset time estimate
2. Estimate the amplitude.
3. Where the station being processed is a sensor array station, generate the f-k spectrum.
4. Where the station being processed is a sensor array station of long baseline dimensions, generate the f-k spectrum via beam.
5. Where the station being processed in a 3-C single-sensor station, estimate polarization.

When all available *detection* objects have been processed within the main loop, a secondary loop is run over defective *detection* objects. Within this secondary loop each defective *detection* object is removed from the input *detection* container. Finally, the revised *detection* input container is returned to the calling function.

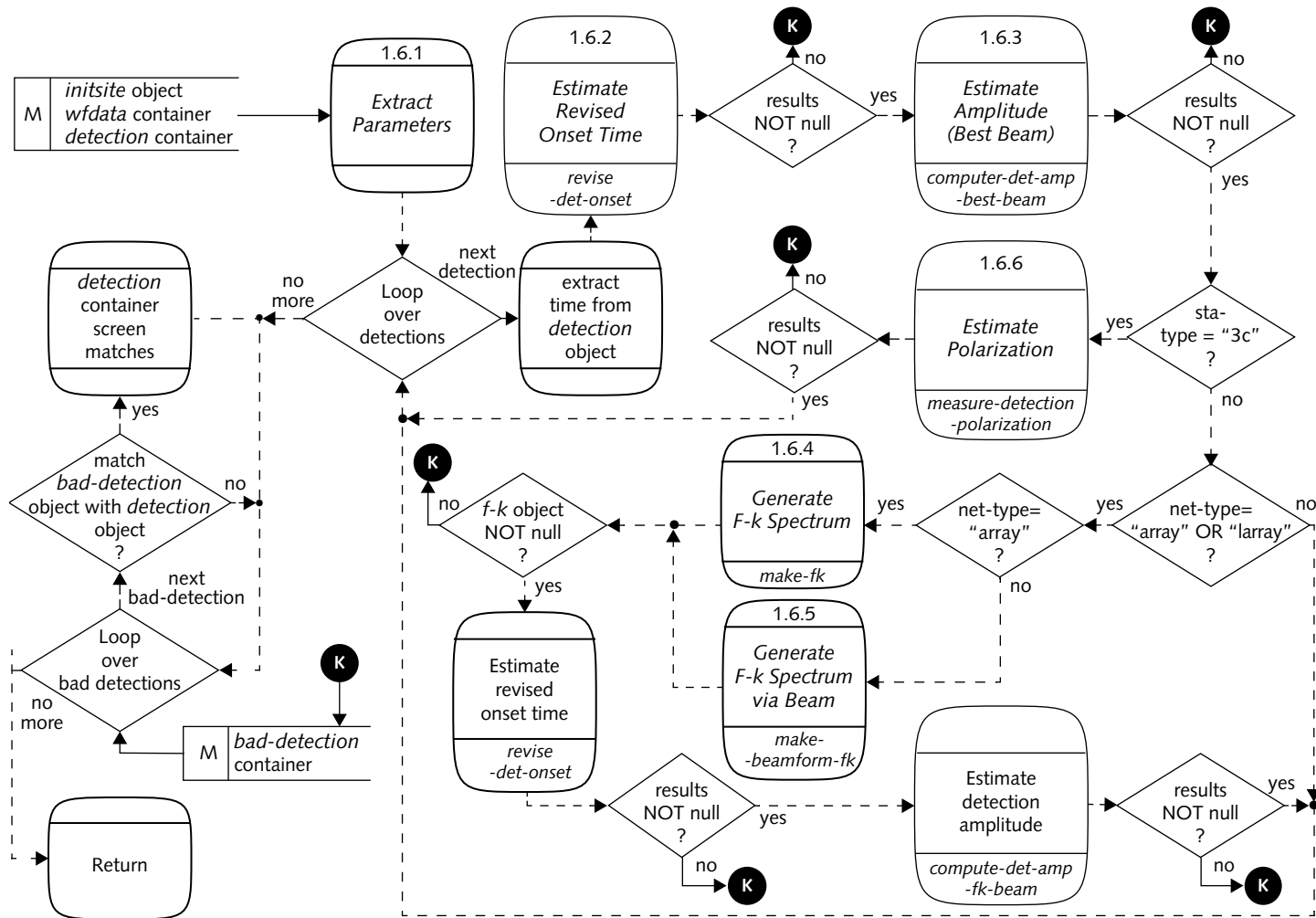


FIGURE 16. ARCHITECTURE OF REVISE EXTRACTED ESTIMATES

## ▼ Seismic Processing

The following subprocesses shown in Figure 16 are described in this section:

- *Extract Parameters* [1.6.1]
- *Estimate Revised Onset Time* [1.6.2]
- *Estimate Amplitude (Best Beam)* [1.6.3]
- *Generate F-k Spectrum* [1.6.4]
- *Generate F-k Spectrum via Beam* [1.6.5]
- *Estimate Polarization* [1.6.6]

### Extract Parameters [1.6.1]

The process *Revise Extracted Estimates* requires a variety of parameter values and symbols to be set for it to operate. The subprocess *Extract Parameters* acquires and binds these parameter values and symbols. It is implemented by the intrinsic and common *libgobj* Scheme functions `let*` and `extract-gobj-attr`, respectively. It accesses the attributes in the *initsite* object and creates the local symbols listed in Table 46.

**TABLE 46: PARAMETERS ACQUIRED BY EXTRACT PARAMETERS**

Parameter Name	Source
<i>det-rec</i>	<i>initsite</i> object
<i>onset-rec</i>	<i>initsite</i> object
<i>SNRfilt-rec</i>	<i>initsite</i> object
<i>amprec-con</i>	<i>initsite</i> object
<i>fk-rec</i>	<i>initsite</i> object
<i>polar-rec</i>	<i>initsite</i> object
<i>use-polar-azi-slow?</i>	<i>initsite</i> object
<i>nettype</i>	<i>initsite</i> object
<i>statype</i>	<i>initsite</i> object
<i>wavetype</i>	<i>initsite</i> object



In addition to creating these local symbols, this function creates an initially empty *detection* container (symbol *bad-det-con*) for detections that are found to be defective.

### Estimate Revised Onset Time [1.6.2]

The process *Revise Extracted Estimates* revises preliminary memory-resident values for some primary detection attributes and provides initial values for some primary detection attributes for the detections generated in *SeisDet*. The subprocess *Estimate Revised Onset Time* specifically revises the onset time estimated for each detected signal. It operates on one detection at a time. It is implemented by the local Scheme function `revise-det-onset`. This function is called by the Scheme language code:

```
(revise-det-onset wfdata-con onset-rec SNRfilt-rec
  beam-type det)
```

The function requires five arguments, which are described in Table 47.

**TABLE 47: FUNCTIONAL INTERFACE OF REVISE-DET-ONSET**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>onset-rec</i>	input	GObj	<i>onset-recipe</i> object
<i>SNRfilt-rec</i>	input	GObj	<i>SNRfilt-recipe</i> object
<i>beam-type</i>	input	character-string	type of beam to process: "fk" or "best"
<i>det</i>	input and output	container	<i>detection</i> objects
n/a	returned value	GObj	<i>detection</i> object nil (indicates error)

## ▼ Seismic Processing

This function is called with the argument *beam-type* set as the character-string “best”. This function revises the onset time estimate using the appropriate algorithm for the input beam type, modifies the input *detection* object, and returns this object. On error, *nil* is returned. If the *ons-SNR-filter* parameter of the *onset-recipe* object is set appropriately, this algorithm searches over the SNRfilter bank to identify and use in subsequent processing within this function the filter that yields the beam with the best snr.

The function *revise-det-onset* performs the following steps:

1. Extract parameters from the *onset-recipe* object and the input *detection* object.
2. Copy the *beam-recipe* object *best-beam-rec* from the input *detection* object.
3. In the case where the input beam type is *fk*, set the *azi* and *slow* attributes of the beam recipe to the default value (−1).
4. Where the *ons-SNR-filter* parameter of the *onset recipe* object is set, search over the SNRfilter bank to identify the beam with the best snr.
5. Where the onset revision method specifies an auto-regressive Akaike Information Criterion (ARAIC) approach, generate and return an *onset-beam* object applying the ARAIC method. Also reset the *time* attribute in the *detection* object to the estimate produced by this ARAIC algorithm.
6. Where the onset revision method specifies a non-AIC approach, generate and return an *onset-beam* object applying the traditional AR method. Also, reset the *time* attribute in the *detection* object to the estimate produced by this traditional AR algorithm.

An error in any of steps 4–6 causes an error message to be printed, and the process returns a value of *nil*.

The onset revision functions referenced in *revise-det-onset* are actually implemented by the *DFX/libonset* functions *revise-onset* and *revise-farm-onset*. Function *revise-det-onset* returns a *detection* object whose *time* attribute has been revised. See [IDC5.2.1] for more information on auto-regressive approaches to refining onset time estimates.

**Estimate Amplitude (Best Beam) [1.6.3]**

The process *Revise Extracted Estimates* revises preliminary memory-resident values for some primary detection attributes and provides initial values for the remaining primary detection attributes for the detections generated in *SeisDet*. The subprocess *Estimate Amplitude (Best Beam)* specifically estimates the amplitude for a given *detection* object using the available waveforms, detection recipe, amplitude recipe container, and *detection* object from the input. It operates on one detection at a time. It is implemented by the global Scheme function `compute-det-amp-best-beam`. This function is called by the Scheme language code:

```
(compute-det-amp-best-beam wfdata-con amprec-con
  det-rec det)
```

The function requires four arguments, which are described in Table 48.

**TABLE 48: FUNCTIONAL INTERFACE OF COMPUTE-DET-AMP-BEST-BEAM**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>amprec-con</i>	input	GObj	<i>amplitude estimation recipe</i> object
<i>det-rec</i>	input object	GObj	<i>seismic detection processing recipe</i> object
<i>det</i>	input and output	GObj	<i>detection</i> object
n/a	returned value	GObj	<i>detection</i> object <i>nil</i> (indicates error)

The *amplitude-estimation recipe* object specified in the *amplitude-estimation recipe* container (`det-amprec-for-arrival`) is used in conjunction with the best beam to generate the amplitude estimate.

## ▼ Seismic Processing

Function `compute-det-amp-best-beam` is implemented in global Scheme code, which provides a “wrapper” to the *DFX/libamp* function `measure-detection-amplitude`. This function returns an *amplitude* object and modifies the *detection* object input to it. The input *detection* object has its *amp* and *per* attributes set in accordance with the corresponding parameters of the returned *amplitude* object. The returned value from `compute-det-amp-best-beam` is this *detection* object in which the *amp* and *per* attributes have been set.

**Generate F-k Spectrum [1.6.4]**

The process *Revise Extracted Estimates* revises preliminary memory-resident values for some primary detection attributes and provides initial values for the remaining primary detection attributes for the detections generated in *SeisDet*. The subprocess *Generate F-k Spectrum* specifically computes the frequency-wavenumber spectrum for the available waveforms in accordance with the *f-k recipe* object in the neighborhood of the detected signal. For a given detection, this spectrum provides estimates of the azimuth and magnitude slowness and estimates of the errors in these two parameters. It operates on one detection at a time. It is implemented by the Scheme function `make-fk`. This function is called by the Scheme language code:

```
(make-fk wfdata-con det fk-rec)
```

The function requires three arguments, which are described in Table 49.

**TABLE 49: FUNCTIONAL INTERFACE OF MAKE-FK**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>det</i>	input and output	GObj	<i>detection</i> object
<i>fk-rec</i>	input	GObj	<i>f-k recipe</i> object
n/a	returned value	GObj	<i>f-k spectrum</i> object <i>nil</i> (indicates error)

The time point at which the f-k spectrum is generated is determined by the time parameter in the input *detection* object. The processing time window is set in accordance with the *f-k recipe's lead* and *lag* parameters. The waveforms are pre-filtered, and the frequency range over which the f-k spectrum is averaged is set by the *flo* and *fhi*, and the *fklof* and *fkhi* parameters of the *f-k recipe* object, respectively. The slowness extent and resolution of the f-k spectrum calculation are likewise set in accordance with the *f-k recipe* object's *maxslow* and *nslow* parameters. The f-k spectrum is scaled to the F-statistic, and the point of maximum F-statistic in vector slowness space is transformed to the back-azimuth and magnitude slowness estimates for the detection. Estimates of the measurement errors, *delaz* and *delslo*, in these parameters are also produced. The estimates are set into both the returned f-k object and the appropriate parameters of the input *detection* object. See [IDC5.2.1] for more information on frequency-wavenumber spectrum generation.

This function is implemented by the *DFX/libfk* function `make-fk`. The returned value from `make-fk` is an f-k (spectrum) object, or `nil` if an error occurred.

### Generate F-k Spectrum via Beam [1.6.5]

The process *Revise Extracted Estimates* revises preliminary memory-resident values for some primary detection attributes and provides initial values for the remaining primary detection attributes for the detections generated in *SeisDet*. The subprocess *Generate F-k Spectrum via Beam* specifically computes the frequency-wavenumber spectrum for the available waveforms in accordance with the *f-k recipe* object in cases where the spatial extent of the sensor array is too large for the normal subprocess *Generate F-k Spectrum*. For a given detection, this spectrum provides estimates of the azimuth and magnitude slowness and estimates of the errors in these two parameters. It operates on one detection at a time. It is implemented by the Scheme function `make-beamform-fk`. This function is called by the Scheme language code:

```
(make-beamform-fk wfdata-con det fk-rec)
```

The function requires the same arguments described in Table 49 on page 128.

## ▼ Seismic Processing

The function `make-beamform-fk` computes the frequency-wavenumber spectrum using time-domain beamforming rather than the frequency-domain approach applied in the normal algorithm. This is essential where long baseline sensor arrays are in use; inter-sensor delays can easily exceed the total time duration of the f-k spectrum processing window (the sum of the *lead* and *lag* attributes in the f-k spectrum recipe). Pre-calculated time delays are used.

The time-point, frequency limits for pre-filtering the data and for the spectrum calculation are set in the same way as described in the preceding subprocess *Generate F-k Spectrum*. The slowness extent and resolution also are set in the same way as previously described. As with the prior subprocess, this f-k spectrum is also scaled to the F-statistic, and the point of maximum F-statistic in vector-slowness space is transformed to the back-azimuth and magnitude slowness estimates for the detection. Estimates of the measurement errors, *delaz* and *delslo*, in these parameters are also produced. The estimates are set in the produced *f-k* object and in the appropriate attributes of the input *detection* object.

This function is implemented by the *DFX/libfk* function `make-beamform-fk`. The returned value from `make-beamform-fk` is an *f-k* (spectrum) object, or `nil` if an error has occurred.

Following the generation of the f-k spectrum by either of the two foregoing subprocesses, the subprocesses *Estimate Revised Onset Time* and *Estimate Detection Amplitude* are invoked in order to improve the onset time estimate and to extract better amplitude and period information now that azimuth and magnitude slowness estimates are available from the f-k spectrum for this detection. These subprocesses call the functions `revise-det-onset` and `compute-det-amp-fk-beam`, respectively. The `revise-det-onset` function operates exactly as described in “Estimate Revised Onset Time [1.6.2]” on page 125. The `compute-det-amp-fk-beam` function operates the same as the `compute-det-amp-best-beam` function described in “Estimate Amplitude (Best Beam) [1.6.3]” on page 127 with the following exception: in `compute-det-amp-best-beam` the beam recipe azimuth and magnitude slowness values govern the computations performed by

`measure-detection-amplitude`, whereas in `compute-det-amp-fk-beam` the *detection* object azimuth and magnitude slowness values govern the computations performed by `measure-detection-amplitude`.

### Estimate Polarization [1.6.6]

The process *Revise Extracted Estimates* revises preliminary memory-resident values for some primary detection attributes and provides initial values for the remaining primary detection attributes for the detections generated in *SeisDet*. Where the waveforms were recorded at stations having 3-C seismic sensors, the subprocess *Estimate Polarization* computes a variety of polarization parameters from the available waveforms in accordance with the polarization recipe, as provided by the *detection* object. This subprocess operates on one detection at a time. It is implemented by the *DFX/libthreec* function `measure-detection-polarization`, which is called by the Scheme language code:

```
(measure-detection-polarization wfdata-con polar-rec det)
```

The function requires three arguments, which are described in Table 50.

**TABLE 50: FUNCTIONAL INTERFACE OF MEASURE-DETECTION-POLARIZATION**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>polar-rec</i>	input	GObj	<i>polarization estimation recipe</i> object
<i>det</i>	input and output	GObj	<i>detection</i> object
n/a	returned value	GObj	<i>polar</i> object nil (indicates error)

The returned value from `measure-detection-polarization` is a *polar* object, which is `nil` if an error occurred.

## ▼ Seismic Processing

In addition, if the polarization recipe parameter *use\_azi\_slow* is set to unity, then the azimuth and magnitude slowness estimates and their estimated errors (*delaz* and *delslo*) from the polarization processing are set into the appropriate parameters of the input *detection* object where they are returned as a side-effect. See [IDC5.2.1] for more information on polarization estimation.

**Detailed Description of Perform  
Feature Extraction [1.8]**

This process is implemented by a call to the local Scheme function *measure-detection-features*. This function performs the processing that provides estimates of the amplitude, sbsnr amplitude, and periods for detections identified by the process *Perform Detection*. All generated information is written to the appropriate database tables: **arrival**, **detection**, **amplitude**, **amp3c**, and **apma**. The types of features estimated depend on the nature of the station.

Figure 17 shows the architecture of *Perform Feature Extraction*. The process first extracts parameters, then it performs a loop over the *detection* objects. Within the loop over *detection* objects, the following steps are applied to each *detection* object:

1. Create and write *dbarrival* object.
2. Where the station has 3-C sensors, generate and write the *dbapma* object.
3. Estimate and write the detection amplitudes.
4. Generate and write detection beams.
5. Create and write the *dbdetection* object.
6. Estimate and write sbsnr amplitudes.
7. Where the station is a single 3-C sensor, estimate and write 3-C amplitudes.



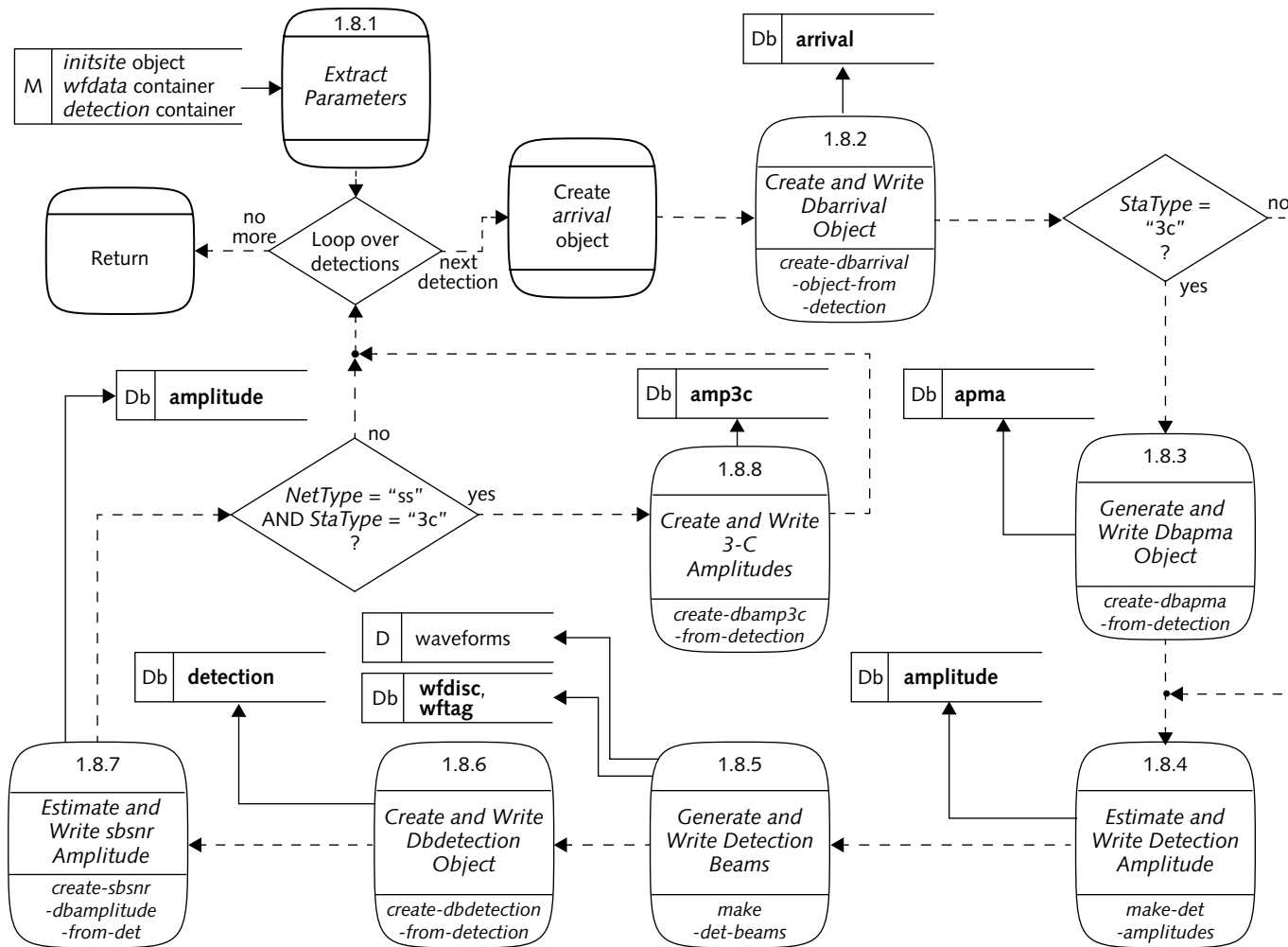


FIGURE 17. ARCHITECTURE OF PERFORM FEATURE EXTRACTION

## ▼ Seismic Processing

When all *detection* objects have been processed within the loop and it is completed, the database inserts are committed. If an error occurs in the *dbarrival* or *dbapma* steps (1 or 2), the current database transactions are rolled-back, and the next detection is processed. If an error occurs in a subsequent step (3–7), a warning message is printed, and processing continues to the next step for the current *detection* object.

The following subprocesses shown in Figure 17 are described in this section:

- *Extract Parameters [1.8.1]*
- *Create and Write Dbarrival Object [1.8.2]*
- *Generate and Write Dbapma Object [1.8.3]*
- *Estimate and Write Detection Amplitude [1.8.4]*
- *Generate and Write Detection Beams [1.8.5]*
- *Create and Write Dbdetection Object [1.8.6]*
- *Estimate and Write sbsnr Amplitude [1.8.7]*
- *Create and Write 3-C Amplitudes [1.8.8]*

### Extract Parameters [1.8.1]

The process *Perform Feature Extraction* requires a variety of parameter values and symbols to be set for it to operate. The subprocess *Extract Parameters* acquires and binds these parameter values and symbols. It is implemented by the intrinsic, global, and common *libgobj* Scheme functions `let*`, `mstspar`, and `extract-gobj-attr`, respectively. It accesses the CVAR datastore directly as well as parameters in the *initsite* object and creates the local symbols listed in Table 51.

**TABLE 51: PARAMETERS ACQUIRED BY EXTRACT PARAMETERS**

Parameter Name	Source
<i>arrival-table</i>	CVAR
<i>detection-table</i>	CVAR
<i>apma-table</i>	CVAR

**TABLE 51: PARAMETERS ACQUIRED BY EXTRACT  
PARAMETERS (CONTINUED)**

Parameter Name	Source
<i>amplitude-table</i>	CVAR
<i>amp3c-table</i>	CVAR
<i>det-rec</i>	<i>initsite</i> object
<i>amprec-con</i>	<i>initsite</i> object
<i>beamtirec-rec</i>	<i>initsite</i> object
<i>polar-rec</i>	<i>initsite</i> object
<i>amp3crec-con</i>	<i>initsite</i> object
<i>stdbeam-con</i>	<i>initsite</i> object
<i>NetType</i>	<i>initsite</i> object
<i>statype</i>	<i>initsite</i> object
<i>wavetype</i>	<i>initsite</i> object
<i>use-polar-azi-slow?</i>	<i>polar-rec</i> object (see local symbol above)

This function also creates two formatted messages: an error message format and a warning message format for use as required.

### Create and Write Dbarrival Object [1.8.2]

The process *Perform Feature Extraction* requires an arrival object to be written to the database. The subprocess *Create and Write Dbarrival Object* creates a *dbarrival* object from the attributes of an existing *detection* object, and it submits this to the **arrival** database table. It is implemented by the *DFX/libdb* function `create-dbar-rival-from-detection` and the global Scheme function `submit-object-db`. The first function is called by the Scheme language code:

```
(create-dbarrival-from-detection det)
```

## ▼ Seismic Processing

The function requires one argument, which is described in Table 52.

**TABLE 52: FUNCTIONAL INTERFACE OF CREATE-DBARRIVAL-FROM-DETECTION**

Argument Name	Usage	Type	Description
<i>det</i>	input	GObj	<i>detection</i> object
n/a	returned value	container	<i>dbarrival</i> objects nil (indicates error)

This function copies matching attributes from the input *detection* object into the *dbarrival* object to the extent possible, and it sets others (*chan*, *auth*, *jdate*, and *lddate*). The bulk of the parameter attribute copying is accomplished by the *DFX/libutil* function *gobj\_to\_gobj\_copy*. Both the estimated magnitude slowness (*slow*) and its estimated error (*delslo*) are converted from seconds per kilometer to seconds per degree. This function returns a *dbarrival* object, or nil, if an error was encountered. On error, an error message is printed, and waveform feature extraction and processing continues with the next detection.

Following the generation of the *dbarrival* object, it is submitted to the database for insertion into the **arrival** table. This function is implemented by the global Scheme function `submit-object-db`, which is called by the Scheme language code:

```
(submit-object-db dbarrival arrival-table)
```

The function requires two arguments, which are described in Table 53.

**TABLE 53: FUNCTIONAL INTERFACE OF SUBMIT-OBJECT-DB (ARRIVAL)**

Argument Name	Usage	Type	Description
<i>dbarrival</i>	input	GObj	<i>dbarrival</i> object
<i>arrival-table</i>	input	character-string	<b>arrival</b> database table
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The function `submit-object-db` inserts the *dbarrival* object into a suitable container (a container of the given object class) and submits the container to the database by the global Scheme function `submit-container-db` for insertion into the **arrival** table. On error, an error message is printed, and a value of `nil` is returned to the calling function. For more information on submitting results to the database see “Writing to the Database” on page 44.

### Generate and Write Dbapma Object [1.8.3]

The process *Perform Feature Extraction* requires a particle motion analysis (apma) object to be written to the database. The subprocess *Create and Write Dbapma Object* creates a *dbapma* object from the input *detection* object, and it submits this to the **apma** database table. It is implemented by the *DFX/libdb* function `create-dbapma-from-detection` and the global Scheme function `submit-object-db`. The first function is called by the Scheme language code:

```
(create-dbapma-from-detection det)
```

The function requires one argument, which is described in Table 54.

**TABLE 54: FUNCTIONAL INTERFACE OF CREATE-DBAPMA-FROM-DETECTION**

Argument Name	Usage	Type	Description
<i>det</i>	input	GObj	<i>detection</i> object
n/a	returned value	GObj	<i>dbapma</i> object <code>nil</code> (indicates error)

This function copies matching attributes from the input *detection* object into the *dbapma* object to the extent possible, and it sets others (*auth* and *lddate*). The bulk of the parameter attribute copying is accomplished by the *DFX/libutil* function `gobj_to_gobj_copy`. This function returns a *dbapma* object, or `nil` if an error is encountered. See [IDC5.2.1] for more information on seismic particle motion analysis.

## ▼ Seismic Processing

Following the generation of the *dbapma* object, it is submitted to the database for insertion into the **apma** table. This function is implemented by the global Scheme function `submit-object-db`, which is called by the Scheme language code:

```
(submit-object-db dbapma apma-table)
```

The function requires two arguments, which are described in Table 55.

**TABLE 55: FUNCTIONAL INTERFACE OF SUBMIT-OBJECT-DB (APMA)**

Argument Name	Usage	Type	Description
<i>dbapma</i>	input	GObj	<i>dbapma</i> object
<i>apma-table</i>	input	character-string	<b>apma</b> database table
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The function `submit-object-db` inserts the *dbapma* object into a suitable container (a container of the given object class) and submits the container to the database via the global Scheme function `submit-container-db` for insertion into the **apma** table. On error, an error message is printed, and a value of `nil` is returned to the calling function. For more information on submitting results to the database see “Writing to the Database” on page 44.

### Estimate and Write Detection Amplitude [1.8.4]

The process *Perform Feature Extraction* requires detection amplitudes to be estimated and written to the database. The subprocess *Estimate and Write Detection Amplitude* performs these measurements, and it submits the results to the **amplitude** database table. It is implemented by the *DFX/libamp* function `make-det-amplitudes` and the global Scheme function `submit-container-db`. The first function is called by the Scheme language code:

```
(make-det-amplitudes wfdata-con amprec-con det)
```

The function requires three arguments, which are described in Table 56.

TABLE 56: FUNCTIONAL INTERFACE OF MAKE-DET-AMPLITUDES

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>amprec-con</i>	input	container	<i>amplitude estimation processing recipe</i> objects
<i>det</i>	input and output	GObj	<i>detection</i> object
n/a	returned value	container	<i>dbamplitude</i> objects

This function estimates the detection amplitudes of the input *detection* object for each of the *detection amplitude recipe* objects in the input container using the waveforms. It returns a *dbamplitude* container, or `nil` if an error is encountered. As a side-effect the *amp* and *per* parameters of the input *detection* object are set with the estimated amplitude and period. On error, a warning message is printed, further processing is attempted, and control is passed to the next subprocess within the process *Perform Feature Extraction*. See [IDC5.2.1] for more information on seismic amplitude estimation.

Following the generation of the *dbamplitude* container, it is submitted to the database for insertion into the **amplitude** table. This function is implemented by the global Scheme function `submit-container-db`, which is called by the Scheme language code:

```
(submit-container-db dbamplitude-con amplitude-table)
```

The function requires two arguments, which are described in Table 57.

## ▼ Seismic Processing

**TABLE 57: FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (AMPLITUDE)**

Argument Name	Usage	Type	Description
<i>dbamplitude-con</i>	input	container	<i>dbamplitude</i> objects
<i>amplitude-table</i>	input	character-string	<b>amplitude</b> database table
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The function `submit-container-db` submits the *dbamplitude* container to the database for insertion into the **amplitude** table. On error, an error message is printed, and control is passed to the next subprocess within the process *Perform Feature Extraction*. For more information on submitting results to the database see “Writing to the Database” on page 44.

### Generate and Write Detection Beams [1.8.5]

The process *Perform Feature Extraction* requires detection beams to be generated and written to the UNIX filesystem and the database. The subprocess *Generate and Write Detection Beams* generates the detection beams, writes the time-series beam data samples to the UNIX filesystem, and submits the corresponding beam *dbwf-disc* record to the **wfdisc** database table and (if parameter *data-output-wftag* is set to non-zero) to the **wftag** database table. It is implemented by the local Scheme functions `make-det-beams` and `submit-dbwf-disc-dbwftag`. The first function is called by the Scheme language code:

```
(make-det-beams wfddata-con beamtirec-con det-rec det)
```

This function requires four arguments, which are described in Table 58.



TABLE 58: FUNCTIONAL INTERFACE OF MAKE-DET-BEAMS

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>beamtirec-con</i>	input	container	<i>beam time-interval-recipe</i> objects
<i>det-rec</i>	input	GObj	<i>seismic detection processing recipe</i> object
<i>det</i>	input and output	GObj	<i>detection</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The function generates detection beams for each *time-interval-recipe* object in the *time-interval-recipe* container. The waveforms used to generate the detection beams are extracted from the input *wfdata* container. The *detection-recipe* object supplies the value of the parameter *det-output-beam-max-slow*, which is compared with the value of *slow* extracted from the input *detection* object to ensure that detection beams are generated only if the criterion on maximum (magnitude) slowness is satisfied. The detection beams are generated by the *DFX/libbio* function `create-db-detection-beam`. This function has two important activities: first, it invokes the *DFX/libbeam* function `gobj_make_beam` to perform the beamforming, and second, it writes the time series of beamformed samples to the UNIX file-system.

The function `create-db-detection-beam` returns a list consisting of a *dbwfdisc* container and a *dbwftag* container. The list is `nil` if an error was encountered.

The containers of *dbwfdisc* and *dbwftag* objects are written to the database by the local Scheme function `submit-dbwfdisc-dbwftag`. This submission function commits the database if the submission is successful; on error it rolls back the database to the state it had before calling this submission. On error, a warning message is printed, further processing is attempted, and control is passed to the next subprocess within the process *Perform Feature Extraction*.

## ▼ Seismic Processing

Each detection beam is written to the UNIX filesystem in the directory and filenames delineated by the *dir* and *dfile* attributes of the corresponding record in the **wfdisc** database table. The directory name is of the form */base/YYYY/JJJ/* where *JJJ* is the three-digit Julian day of the year (with lead zeros as required), and *YYYY* is the four-digit year for the given input waveforms used to generate the detection beam. The base directory path is set in the **DFX-detection.par** file by the *data-output-base-directory* parameter.

### Create and Write Dbdetection Object [1.8.6]

The process *Perform Feature Extraction* requires *detection* objects to be created and written to the database. The subprocess *Create and Write Dbdetection Object* creates the *dbdetection* objects, and it submits these to the **detection** database table. It is implemented by the **DFX/libdb** function `create-dbdetection-from-detection` and the global Scheme function `submit-object-db`. The first function is called by the Scheme language code:

```
(create-dbdetection-from-detection det)
```

The function requires one argument, which is described in Table 59.

**TABLE 59: FUNCTIONAL INTERFACE OF CREATE-DBDETECTION-FROM-DETECTION**

Argument Name	Usage	Type	Description
<i>det</i>	input	GObj	<i>detection</i> object
n/a	returned value	GObj	<i>dbdetection</i> object nil (indicates error)

This function copies matching attributes from the input *detection* object into the *dbdetection* object to the extent possible, and it sets others (*chan*, *bmtime*, *jdate*, and *lddate*). The bulk of the parameter attributes copying is accomplished by the **DFX/libutil** function `gobj_to_gobj_copy`. This function returns a *dbdetection*

object, or `nil` if an error was encountered. On error, a warning message is printed, further processing is attempted, and control is passed to the next subprocess within the process *Perform Feature Extraction*.

Following the generation of the *dbdetection* object, it is submitted to the database for insertion into the **detection** table. This function is implemented by the global Scheme function `submit-object-db`, which is called by the Scheme language code:

```
(submit-object-db dbdetection detection-table)
```

The function requires two arguments, which are described in Table 60.

**TABLE 60: FUNCTIONAL INTERFACE OF SUBMIT-OBJECT-DB (DETECTION)**

Argument Name	Usage	Type	Description
<i>dbdetection</i>	input	GObj	<i>dbdetection</i> object
<i>detection-table</i>	input	character-string	<b>detection</b> database table
n/a	returned value	logical scalar	indicates success ( <code>t</code> ) or failure ( <code>nil</code> )

The function `submit-object-db` inserts the given object into a suitable container (a container of the given object class) and submits the container to the database by the global Scheme function `submit-container-db` for insertion into the **detection** table. On error, an error message is printed, no further processing is accomplished, and control is returned to the calling function. For more information on submitting results to the database see “Writing to the Database” on page 44.

## ▼ Seismic Processing

**Estimate and Write sbsnr  
Amplitude [1.8.7]**

The process *Perform Feature Extraction* requires sbsnr amplitudes to be estimated and written to the database. The subprocess *Estimate and Write sbsnr Amplitude* performs these measurements, and it submits the results to the **amplitude** database table. It is implemented by the global function `create-sbsnr-dbamplitude-from-det` and the global Scheme function `submit-container-db`. The first function is called by the Scheme language code:

```
(set! dbsbsnr-con (create-sbsnr-dbamplitude-from-det
  stdbeam-con det))
```

The function requires two arguments, which are described in Table 61.

**TABLE 61: FUNCTIONAL INTERFACE OF CREATE-SBSNR-DBAMPLITUDE-FROM-DET**

Argument Name	Usage	Type	Description
<i>stdbeam-con</i>	input	container	<i>beam</i> objects
<i>det</i>	input	GObj	<i>detection</i> object
n/a	returned value	container	<i>dbamplitude</i> objects <i>nil</i> (indicates error)

This function estimates the sbsnr amplitude of the input *detection* object for each of the standard beams in the input container. The sbsnr amplitude is a measure based on the STAV and LTAV measures from a standard beam at the time of the detection. This function returns a *dbamplitude* container (symbol *dbsbsnr-con*), or *nil* if an error was encountered. On error, a warning message is printed, further processing is attempted, and control is passed to the next subprocess within the process *Perform Feature Extraction*. See [IDC5.2.1] for more information on seismic amplitude estimation.

Following the generation of the *dbamplitude* container (symbol *dbbsnr-con*), it is submitted to the database for insertion into the **amplitude** table. This function is implemented by the global Scheme function `submit-container-db`, which is called by the Scheme language code:

```
(submit-container-db dbbsnr-con amplitude-table)
```

The function requires two arguments, which are described in Table 62.

**TABLE 62: FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (SBSNR)**

Argument Name	Usage	Type	Description
<i>dbbsnr-con</i>	input	container	<i>dbamplitude</i> objects
<i>amplitude-table</i>	input	character-string	<b>amplitude</b> database table
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The function `submit-container-db` submits the *dbamplitude* container (symbol *dbbsnr*) to the database for insertion into the **amplitude** table. On error, a warning message is printed, further processing is attempted, and control is passed to the next subprocess within the process *Perform Feature Extraction*. For more information on submitting results to the database see “Writing to the Database” on page 44.

### Create and Write 3-C Amplitudes [1.8.8]

The process *Perform Feature Extraction* requires amplitudes to be estimated and written to the database in cases where seismic stations have only a single 3-C sensor. The subprocess *Create and Write Three-component Amplitudes* performs these measurements, and it submits the results to the **amp3c** database table. For the input *detection* object and for each recipe object in the *amp3c recipe* container this function estimates 3-C amplitude using the waveforms in the *wfdata* container.

## ▼ Seismic Processing

This subprocess is implemented by the *DFX/libdb* function `create-dbamp3c-from-detection` and the global Scheme function `submit-container-db`. The first function is called by the Scheme language code:

```
(set! dbamp3c-con (create-dbamp3c-from-detection det
  wfdata-con amp3crec-con))
```

The function requires three arguments, which are described in Table 63.

**TABLE 63: FUNCTIONAL INTERFACE OF CREATE-DBAMP3C-FROM-DETECTION**

Argument Name	Usage	Type	Description
<i>det</i>	input	GObj	<i>detection</i> object
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>amp3crec-con</i>	input	container	<i>3-C amplitude estimation recipe</i> objects
n/a	returned value	container	<i>dbamp3c</i> objects

The function `create-dbamp3c-from-detection` is implemented by the *DFX/libthreec* function `gobj_measure_amp3c` to actually perform the amplitude estimation from 3-C station's waveforms. The function `gobj_measure_amp3c` returns a *dbamp3c* object, or `nil` if an error was encountered. On error, a warning message is printed, further processing is attempted, and control is passed to the next subprocess in the process *Perform Feature Extraction*. For each 3-C amplitude-estimation recipe in the *amp3c* recipe container, the function `gobj_measure_amp3c` is invoked to perform the measurement. The function `create-dbamp3c-from-detection` inserts the *amp3c* objects into a *dbamp3c* container, which it returns. See [IDC5.2.1] for more information on seismic amplitude estimation from 3-C stations.

Following the generation of the *dbamp3c* container (symbol *dbamp3c-con*), it is submitted to the database for insertion into the **amp3c** table. This function is implemented by the global Scheme function `submit-container-db`, which is called by the Scheme language code:

```
(submit-container-db dbamp3c-con amp3c-table)
```

The function requires two arguments, which are described in Table 64.

**TABLE 64: FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (AMP3C)**

Argument Name	Usage	Type	Description
<i>dbamp3c-con</i>	input	container	<i>dbamp3c</i> objects
<i>amp3c-table</i>	input	character-string	<b>amp3c</b> database table
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The function `submit-container-db` submits the *dbamp3c* container to the database for insertion into the **amp3c** table. On error, a warning message is printed, further processing is attempted, and control is passed to the next subprocess in the process *Perform Feature Extraction*. For more information on submitting results to the database see “Writing to the Database” on page 44.

## AUTOMATIC ORIGIN BEAM [2]

*Automatic Origin Beam (OriginBeam)* is a DFX Scheme application that generates beams based on origins for the signals received at the seismic stations. The source code resides in the file `DFX-originbeam.scm`. *OriginBeam* is usually run by specifying the network, start-time, end-time, and the `DFX-originbeam.par` file on the command line. In operations this is governed by a Tuxedo shell. The results of *OriginBeam* are written to the **wfdisc** and **wftag** tables as well as to the UNIX file-system.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Generate and Write Origin Beams* [2.3]

### General Description

*OriginBeam* generates and saves origin beams for the origins in a given time interval and stations in a specified network. It accomplishes this by: (i) identifying the origins, (ii) defining the appropriate time-intervals of waveforms to acquire from each station for all the stations in the specified network, (iii) acquiring and checking the quality of these waveforms, (iv) generating the origin beams steered to the vector slowness of the origin (for the given station), and (v) saving these origin beams in the database and UNIX filesystem.



## Architecture

*OriginBeam* is implemented by a single pass through a sequence of two initialization processes followed by a pair of nested loops over the principal process, as shown in Figure 18. The first initialization process, *Query for Origins*, acquires origin information for the given time interval (2.1). This process is implemented by the function `query-for-db-origin-by-time`, which fills a *dborigin* container with information for all the origins within the given processing time interval. The second initialization process, *Initialize Sites*, acquires for the sites referenced in the origins, all the site-specific information (2.2). This process is implemented by the function `initialize-net-sites`. The third process, *Generate and Write Origin Beams* (2.3) is embedded in the nested loops over origins (outer loop) and sites (inner loop). This process accomplishes the principal objectives of this application: the generation and storage of origin beams to the database and the UNIX filesystem. *Generate and Write Origin Beams* is implemented by the Scheme function `process-origin-for-initsite`.

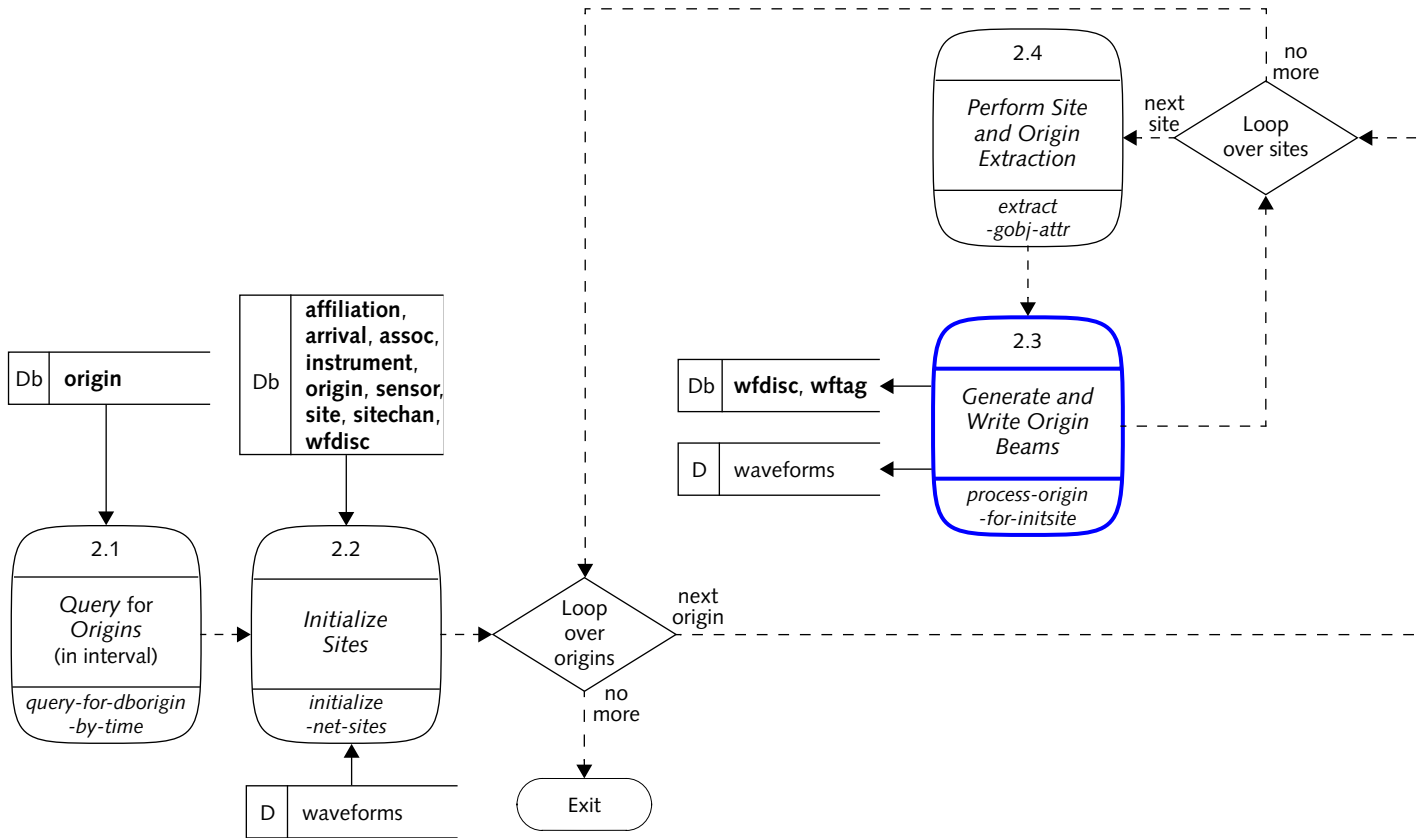


FIGURE 18. ARCHITECTURE OF ORIGINBEAM

## Input/Output

Table 65 lists the input CVAR datastore parameters used in the Scheme code.

**TABLE 65: INPUT PARAMETERS FOR AUTOMATIC ORIGIN BEAM PROCESSING**

Par Name	Source	Default Value	Description
<i>net</i>	command line	<none>	network of stations to process
<i>start-time</i>	command line	<none>	start-time of the processing interval
<i>end-time</i>	command line	<none>	end-time of the processing interval
<i>data-output-wftag</i>	par file	0	flag for <b>wftag</b> table output: 0 means do not write to <b>wftag</b> table
<i>origin-lookback</i>	par file	<none>	time to subtract from both the start- and end-times of the processing interval in order to determine the interval in which to search for origins to process
<i>originbeam-data-offset-len</i>	par file	<none>	time duration of data to add to the ends of the waveforms interval to allow for filter startup and taper-off
<i>originbeam-site-recipe-file</i>	par file	<none>	filename of the site-specific origin beam .par file
<i>wfdisc-extension-len</i>	par file	86400	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX file-system
<i>affiliation-table</i>	par file	<b>affiliation</b>	table containing station and network affiliations

## ▼ Seismic Processing

**TABLE 65: INPUT PARAMETERS FOR AUTOMATIC ORIGIN BEAM PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>instrument-table</i>	par file	instrument	table containing generic (default) calibration information about a station
<i>origin-table</i>	par file	origin	database table containing the origin information
<i>sensor-table</i>	par file	sensor	table containing specific calibration information for physical channels
<i>site-table</i>	par file	site	table containing site-location information
<i>sitechann-table</i>	par file	sitechann	table containing station-channel information
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information
<i>out-wfdisc-table</i>	par file	wfdisc	output table containing waveform file header and descriptive information generated by this application
<i>out-wftag-table</i>	par file	wftag	output table containing waveform mapping information generated by this application

All results from this application are written to the database tables described in Table 66.

**TABLE 66: DATA PRODUCED BY AUTOMATIC ORIGIN BEAM PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>wfdisc</b>	database table	origin beams generated
<b>wftag</b>	database table	maps origins to beams
“.w”	binary files	waveforms origin-beam time-series samples

In addition to the **wfdisc** and **wftag** records in the database, the time series for each origin beam is written to the UNIX filesystem in the directory and filenames delineated by the *dir* and *dfile* attributes of the **wfdisc** table. The directory name is of form */base/YYYY/JJJ/* where *JJJ* is the three-digit Julian day of the year (with lead zeros as required), and *YYYY* is the four-digit year for the given input waveforms used to generate the origin beam. The base directory path is set in the `DFX-originbeam.par` file by the *data-output-base-directory* parameter. This application returns an exit status code whose value indicates whether or not the processing was successful. Successful processing exits with returned value 0; fatal errors have return value 1; if there are no new detections the return value is 2; if there was an unsuccessful retry of an error the returned value is 3.

The **wftag** table is written only if the symbol *data-output-wftag* is a non-zero value.

Table 231 on page 445 lists the specific attributes written by *OriginBeam*.

## Processes

The following processes shown in Figure 18 on page 150 are described in this section:

- *Query for Origins [2.1]*
- *Initialize Sites [2.2]*
- *Generate and Write Origin Beams [2.3]*

## ▼ Seismic Processing

**Query for Origins [2.1]**

This process queries the database for all the origins that occurred during a given time interval. This process is implemented by a call to the local Scheme function `query-for-dborigin-by-time`. It constructs an SQL query, which is submitted to the database. If the query is successful, the results are returned in a *dborigin* container.

This function is called by the Scheme language code:

```
(set! dborigin-con (query-for-dborigin-by-time t1 t2
  origin-table))
```

The function requires three arguments, which are described in Table 67.

**TABLE 67: FUNCTIONAL INTERFACE OF QUERY-FOR-DBORIGIN-BY-TIME**

Argument Name	Usage	Type	Description
<i>t1</i>	input	real	start-time of the interval
<i>t2</i>	input	real	end-time of the interval
<i>origin-table</i>	input	character-string	<b>origin</b> database table from the CVAR datastore
n/a	returned value	container	<i>dborigin</i> objects

For general information on how database queries are implemented see “Querying the Database” on page 41. The function `query-for-dborigin-by-time` returns a *dborigin* container (see the **origin** table description in [IDC5.1.1Rev2]).

### Initialize Sites [2.2]

This process acquires the site-specific information that will be used to calculate time intervals, acquire and check the quality of waveforms, and generate and save origin beams. It is implemented by the local Scheme function `initialize-net-sites`. The function is called by the Scheme language code:

```
(set! initsite-con (initialize-net-sites
  dborigin-container))
```

The function requires one argument, which is described in Table 68.

**TABLE 68: FUNCTIONAL INTERFACE OF INITIALIZE-NET-SITES**

Argument Name	Usage	Type	Description
<i>dborigin-container</i>	input	container	<i>dborigin</i> objects
n/a	returned value	container	<i>initsite</i> objects

This function is described in “Network Initialization” on page 54. The function also initializes the travel-time tables for all seismic phases and sensor stations via the local Scheme function `initialize-net-tt-tables` and establishes the waveforms processing time interval via the local Scheme function `compute-net-ti`.

The function `initialize-net-sites` returns an *initsite* container. The contents of the *initsite* objects for this application are listed in the OriginBeam column of Table 8 on page 50.

### Generate and Write Origin Beams [2.3]

This process performs the principal activities required to generate and save origin beams. This process is implemented by the local Scheme function `process-origin-for-initsite`. The function is called by the Scheme language code:

```
(process-origin-for-initsite initsite dborigin)
```

The function requires two arguments, which are described in Table 69.

## ▼ Seismic Processing

**TABLE 69: FUNCTIONAL INTERFACE OF PROCESS-ORIGIN-FOR-INITSITE**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

For the given site and origin this function performs the following steps:

1. Extract parameters.
2. Calculate the time interval.
3. Acquire and check the quality of the waveforms.
4. Generate the origin beams.
5. Write the beams.

For further information about this function see “Detailed Description of Generate and Write Origin Beams [2.3].”

### **Detailed Description of Generate and Write Origin Beams [2.3]**

This process calculates the required time intervals, acquires and checks the quality of waveforms, generates the origin beams for these time intervals, and saves the beams. It is implemented by the local Scheme function `process-origin-for-initsite`. Figure 19 shows the architecture of this process. It consists of a single pass through several subprocesses. First, the process extracts the variety of parameters needed to generate and write the origin beams. Then it identifies the ti-recipe that satisfies the station-to-event distance criteria. It acquires and checks the quality of waveforms for the time interval required in the ti-recipe. Finally, it generates the beams and writes them to the database.



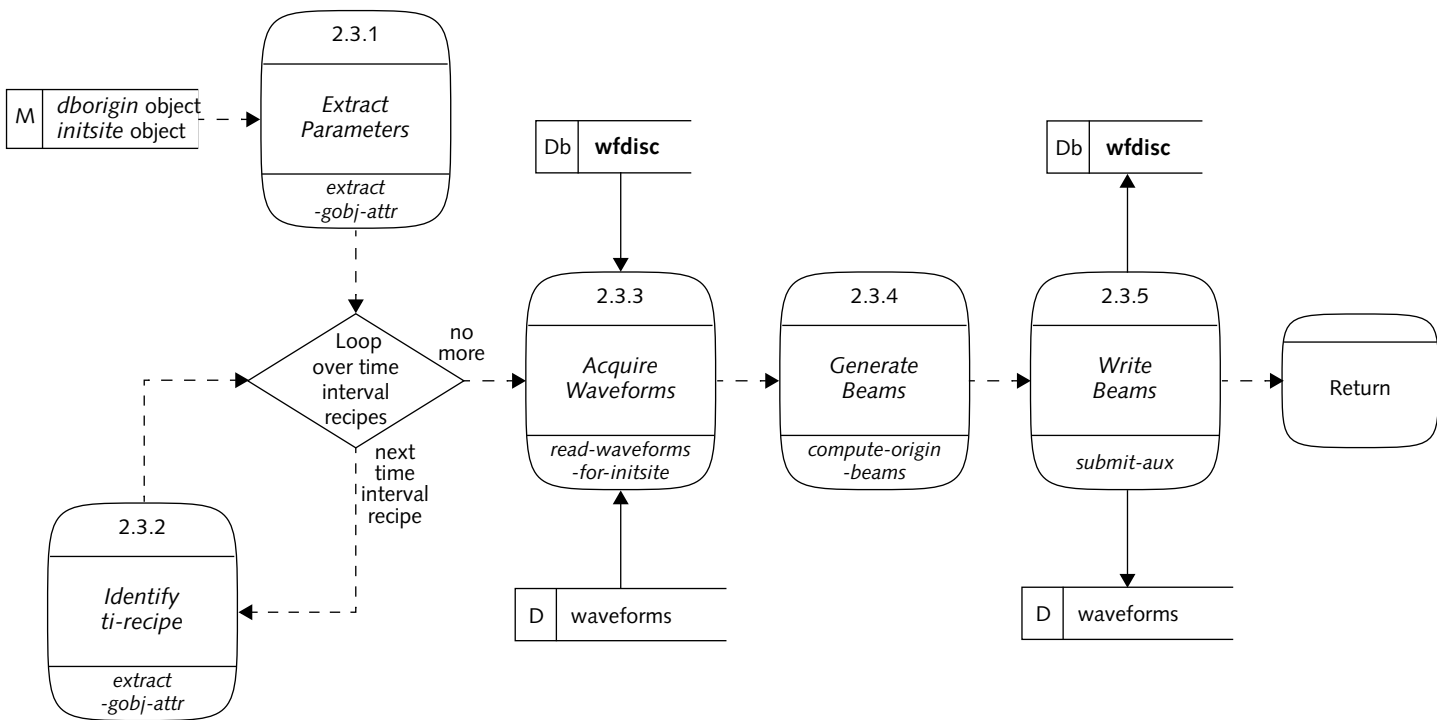


FIGURE 19. ARCHITECTURE OF GENERATE AND WRITE ORIGIN BEAMS

## ▼ Seismic Processing

The following subprocesses shown in Figure 19 are described in this section:

- *Extract Parameters [2.3.1]*
- *Identify ti-recipe [2.3.2]*
- *Acquire Waveforms [2.3.3]*
- *Generate Beams [2.3.4]*
- *Write Beams [2.3.5]*

### Extract Parameters [2.3.1]

This subprocess obtains the parameters needed to generate and write the origin beams. Local symbols are bound through the `let*` macro to parameter values obtained from input arguments. Values are extracted from objects using the common *libgobj* Scheme function `extract-gobj-attr`. In addition, the distance from the station to the event is obtained from the *DFX/libutil* function `compute-distance-azimuth`. Table 70 describes the symbols that are extracted from the *initsite* and *dborigin* objects.

**TABLE 70: PARAMETERS EXTRACTED FROM INITSITE AND DBORIGIN OBJECTS BY PROCESS-ORIGIN-FOR-INITSITE**

Symbol	Type	Source	Description
<i>orid</i>	integer	<i>dborigin</i> object	unique identification number for the origin being processed
<i>dbsite</i>	<i>dbsite</i> object	<i>initsite</i> object	information from the <b>site</b> table
<i>sta</i>	character-string	<i>dbsite</i> object	station code
<i>tirec-con</i>	<i>ti-recipe</i> container	<i>initsite</i> object	recipes used to determine the time interval of the origin beam
<i>beamrec-con</i>	<i>beam recipe</i> container	<i>initsite</i> object	recipes used to determine the nature of the beams to be generated: type, steering information, threshold, filter, and so on

In addition, several other symbols are created in this subprocess. These symbols are described in Table 71.

TABLE 71: INTERNAL PARAMETERS FOR PROCESS-ORIGIN-FOR-INITSITE

Symbol	Type	Source	Description
<i>delta</i>	real	calculated locally by <i>DFX/libutil</i>	station-to-event distance (degrees)
<i>wfdata-con</i>	container <i>wfdata</i> objects	created locally	information about waveforms
<i>dbwfdisc-dbwftag-list</i>	list	created locally	list of <i>dbwfdisc</i> and <i>dbwftag</i> containers for the origin beams that are generated
<i>beam-tirec</i>	<i>time-interval</i> <i>recipe</i> object	created locally	<i>ti-recipe</i> object for this beam

**Identify ti-recipe [2.3.2]**

The process *Generate and Write Origin Beams* computes the origin beam for a station-specific time interval that encompasses the signal arrivals that correspond to the given origin. The subprocess *Identify ti-recipe* obtains the ti-recipe that is used to define this time interval. The global, common *libgobj*, and inline intrinsic Scheme functions *map-container*, *extract-gobj-attr*, and *set!*, respectively, and the *lambda* structure are employed to loop over all the ti-recipes in the *ti-recipes* container (symbol *tirec-con* from Table 70 on page 158). They find the last ti-recipe for which the station-to-event distance (symbol *delta*) lies inside the distance interval specified by the limits *ti-rmin* and *ti-rmax* (which are defined in each ti-recipe). The local symbol *tirec* is set to this recipe. If this function fails to find a ti-recipe that contains the value of *delta* between its limits, the function prints an error message and returns an error status to the calling program.

**Acquire Waveforms [2.3.3]**

This subprocess acquires the appropriate time intervals of waveforms needed to generate and write the origin beams. Quality-control processing is applied to the waveforms during the acquisition process. The local Scheme function *read-wave-*

## ▼ Seismic Processing

`forms-for-initsite` is used to acquire and check the quality of waveforms for the time interval required in the `ti-recipe`. The function is called by the Scheme language code:

```
(read-waveforms-for-initsite initsite dborigin)
```

The function requires two arguments, which are described in Table 72.

**TABLE 72: FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-INITSITE**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
n/a	returned value	container	<i>wfdata</i> objects after quality-control processing

This function returns a *wfdata* container, whose associated waveforms have been checked for quality, for the stations and channels defined by the *initsite* for the time interval indicated by the origin. The waveforms are acquired by a variation of the standard method in *DFX* (see “Reading Waveforms” on page 58). The specific variations are as follows:

- The local symbol *edge-len* (numerically equal in value to *originbeam-data-offset-len*) is extracted from the CVAR datastore by the Scheme function `mstpar`.
- The time interval ( $t1$ ,  $t2$ ) for waveform acquisition is generated by the *DFX/libutil* function `compute-site-origin-ti` from the *dbsite* object (contained within the *initsite* object), the *dborigin* object, and the container of *ti-recipes*.
- This interval ( $t1$ ,  $t2$ ) is adjusted by *edge-len* to eliminate edge effect from filtering and beam steering realignments from the generated beam ( $t1$ ,  $t2$  becomes  $t1-edge-len$ ,  $t2+edge-len$ ).

- The waveforms are checked for quality by the *DFX/libqc* function `qc-waveforms` using the quality-control recipe extracted from the *initsite* object.

### Generate Beams [2.3.4]

This subprocess produces the beams from the waveforms. The local Scheme function `compute-origin-beams` is used to produce the origin beams for this processing interval, origin, and station. It is called by the Scheme language code:

```
(set! dbwfdisc-dbwftag-list (compute-origin-beams
  wfdata-con dbsite dborigin beamrec-con beam-tirec))
```

The function requires five arguments, which are described in Table 73.

**TABLE 73: FUNCTIONAL INTERFACE OF COMPUTE-ORIGIN-BEAMS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>beamrec-con</i>	input	container	<i>beam recipe</i> objects
<i>beam-tirec</i>	input	GObj	<i>time-interval recipe</i> object
n/a	returned value	list	<i>dbwfdisc</i> and <i>dbwftag</i> containers

The function `compute-origin-beams` returns a list consisting of a *dbwfdisc* container and a *dbwftag* container.

The function `compute-origin-beams` performs the following steps:

1. Determine the number of *beam recipe* objects, and acquire this number of unique *wfids* (waveform identity numbers) retaining the maximum *wfid* value as the symbol *max\_wfid*.

## ▼ Seismic Processing

2. Fill the *dbwfdisc* container with the *beam* objects returned by the *DFX/libio* function `create-db-origin-beam`.
3. Return a list consisting of the *dbwfdisc* container and the *dbwftag* container. The *dbwftag* container is an optional return value controlled by the parameter *data-output-wftag* (which must be non-zero in value to generate wftags).

The *DFX/libio* function `create-db-origin-beam` performs the following steps:

1. For each beam recipe in the input *beam recipe* container, compute and write a beam encompassing a time interval determined by the input origin.
2. Write the “.w” files for the time-series data samples of each origin beam to the appropriate file and directory.
3. Create a *dbwfdisc* object for each beam written, and store these in the *dbwfdisc* output container.
4. If required by the value of *data-output-wftag*, create a *dbwftag* object corresponding to each wfdisc object, and store these in the *dbwftag* output container.
5. Return a list consisting of a beam *dbwfdisc* container and a *dbwftag* container (which may be empty). On error, an error message is printed, processing stops, and function `create-db-origin-beam` returns nil.

### Write Beams [2.3.5]

This subprocess saves the generated origin beams in the database and the UNIX filesystem. After the origin beams are generated, the returned *dbwfdisc-dbwftag*-list contains *dbwfdisc* and *dbwftag* containers. The local Scheme function `submit-aux` writes these to the **wfdisc** and **wftag** database tables, respectively. The function is called by the Scheme language code:

```
(submit-aux dbYcon table_name)
```

This function has two arguments, which are described in Table 74.

TABLE 74: FUNCTIONAL INTERFACE OF SUBMIT-AUX

Argument Name	Usage	Type	Description
<i>dbYcon</i>	input	container	<i>dbY</i> objects either <i>dbwfdisc</i> or <i>dbwftag</i> objects
<i>table_name</i>	input	character-string	database table name
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The *out-wfdisc-table* and *out-wftag-table* names are extracted from the CVAR datastore. Inline Scheme code using intrinsic functions `car` and `cadr` is used to extract the *dbwfdisc* and *dbwftag* containers, respectively, from *dbwfdisc-dbwftag-list*. The function `submit-aux` returns `t` or `nil` depending upon whether its processing was successful or not. The function `submit-aux` is a wrapper to the global Scheme function `submit-container-db`. For more information about submitting results to the database see “Writing to the Database” on page 44.

Following this process, the updates to the database are “committed” by the global Scheme function `commit-db`. Also, the “garbage collection” process internal to Scheme is applied explicitly to the local *wfdata* container as well as to their corresponding masks by the global Scheme function `gc-wfdata-container` applied to symbol *wfdata-con*.

## INTERACTIVE BEAM-ON-THE-FLY [3]

*Interactive Beam on the Fly (BOTF)* is a *DFX* Scheme application that computes origin beams for a given origin and phase at a given set of stations. The application source code resides in the file `DFX-botf.scm`. *BOTF* is typically called from within *ARS*. *ARS* sends an IPC message via the *DACS* containing the origin information (latitude, longitude, depth, and time), a list of stations, and the phase. *BOTF* then computes and writes the beams to “.w” files of the UNIX filesystem in the appropriate directory and inserts the corresponding **wfdisc** records into the database.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description covers the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Initialize Sites* [3.2]
- Detailed Description of *Generate Beam* [3.3]

### General Description

*BOTF* was designed to allow the analyst to create origin beams after adding or modifying an origin in *ARS*. A beam is essentially a “delay and sum” operation over a set of waveforms from an array of seismic sensors. *BOTF* predicts the time delays between elements from the origin, station location, and theoretical travel-time tables for the desired phase. It applies the delays to each array element and then sums the shifted waveforms. Because the delays are based on the origin location, the beams produced are referred to as origin beams.



## Architecture

*BOTF* is constructed from three general processes (see Figure 20). The first process extracts the input parameters that define the data to beamform (3.1). The next process retrieves site-specific data and recipe parameters (3.2). The third process does the bulk of the work for this application; it acquires and beamforms the waveforms and then saves the beams to the UNIX filesystem (3.3). It occurs within a loop over stations.

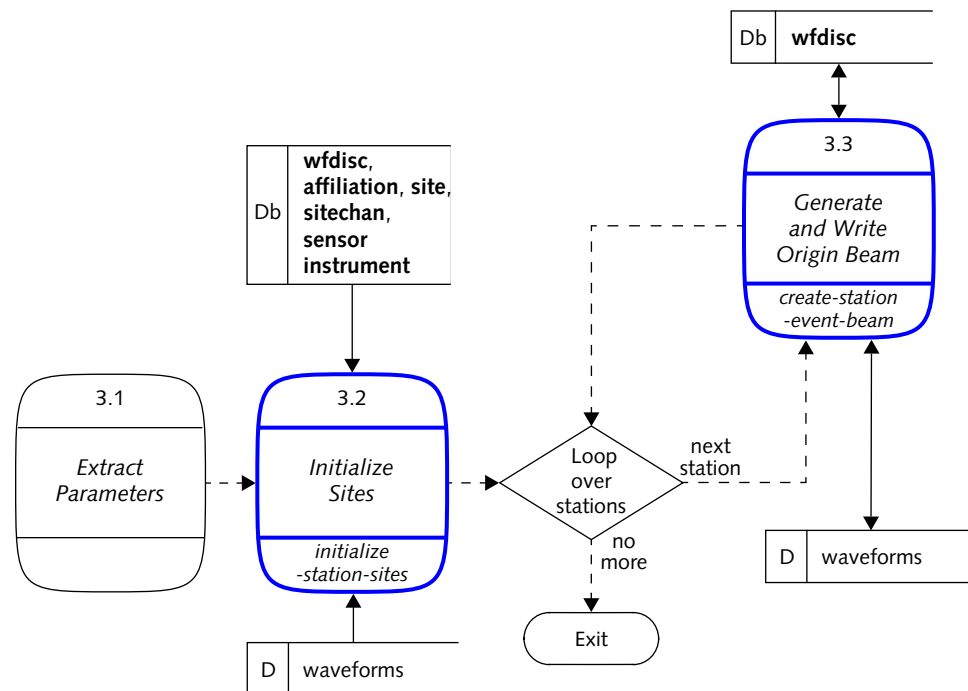


FIGURE 20. ARCHITECTURE OF BOTF

## ▼ Seismic Processing

**Input/Output**

*BOTF* requires a variety of parameters, which are described in Table 75. These parameters are acquired through the CVAR datastore and can be specified either on the command line or in par files. The table indicates the typical method used to specify each parameter. The tables read from the database are described in Table 229 on page 443.

**TABLE 75: INPUT PARAMETERS FOR INTERACTIVE BEAM-ON-THE-FLY PROCESSING**

Par Name	Source	Default Value	Description
<i>database-account</i>	command line	<none>	database access character-string
<i>lat</i>	command line	<none>	origin latitude
<i>lon</i>	command line	<none>	origin longitude
<i>depth</i>	command line	<none>	origin depth
<i>time</i>	command line	<none>	origin time
<i>dir</i>	command line	<none>	UNIX directory for beam output ".w" files
<i>wfdisc_tbl</i>	command line	<none>	<b>wfdisc</b> table for output beams
<i>phase</i> <sup>1</sup>	command line	–	phase for beam calculation
<i>sta</i>	command line	<none>	station code
<i>qc_sta</i>	command line	" "	list of stations on which to check the data quality
<i>excluded_chans</i>	command line	" "	list of array channels to omit from beams
<i>data-offset-len</i>	par file	50.0	time added to either side of calculated time interval when generating beam

**TABLE 75: INPUT PARAMETERS FOR INTERACTIVE BEAM-ON-THE-FLY PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>wfdisc-extension-len</i>	par file	86400	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>botf-tele-beamrec-list</i>	par file	<none>	list of beam names for teleseismic phases
<i>botf-reg-beamrec-list</i>	par file	<none>	list of beam names for regional phases
<i>botf-tele-tirec-list</i>	par file	<none>	list of time-interval recipes for teleseismic phases
<i>botf-reg-tirec-list</i>	par file	<none>	list of time-interval recipes for regional phases
<i>botf-regional-cutoff</i>	par file	20 degrees	regional distance cutoff
<i>botf-site-recipe-file</i>	par file	<none>	par file containing site-specific recipes
<i>affiliation-table</i>	par file	affiliation	table containing station and network affiliations
<i>instrument-table</i>	par file	instrument	table containing generic (default) calibration information about a station
<i>sensor-table</i>	par file	sensor	table containing specific calibration information for physical channels
<i>site-table</i>	par file	site	table containing site-location information
<i>sitechann-table</i>	par file	sitechann	table containing station-channel information
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information

1. The parameter *phase*, if set, overrides the phase types in the beam and time-interval recipes.

## ▼ Seismic Processing

The data produced by *BOTF* is described in Table 76. The beams formed by the application are written to the UNIX filesystem as binary data files. The filenames for the beams are a concatenation of the database account parameter and the unique waveform identification number (*wfid*) with the extension “.w”. They are placed in a subdirectory */base/YYYY/JJJ/* where *base* is the directory specified by the parameter *dir*, *YYYY* is the year, and *JJJ* is the day of year. Database records that point to these binary data files are inserted into the **wfdisc** table specified in the parameter *wfdisc\_tbl*. The exact attributes written are described in Table 232 on page 446.

**TABLE 76: DATA PRODUCED BY INTERACTIVE BEAM-ON-THE-FLY PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>wfdisc</b>	database table	origin beams generated
“.w”	binary files	waveforms origin-beam time-series samples

## Processes

The following processes shown in Figure 20 on page 165 are described in this section:

- *Extract Parameters [3.1]*
- *Initialize Sites [3.2]*
- *Generate and Write Origin Beam [3.3]*

### Extract Parameters [3.1]

This process extracts the input parameters from the CVAR datastore. The parameters extracted define the stations to beamform, the origin used to predict the appropriate time delays, and the phase type (optional) to use. This process is implemented by inline Scheme code. Two primary objects that are specific to *BOTF*

are populated at this stage: The origin information (*lat*, *lon*, *depth*, *time*, *orid*) is placed in a *dborigin* object (symbol *origin*), and the station names are placed in a list (symbol *sta-list*).

### Initialize Sites [3.2]

This process obtains the site-specific information from the database and loads the recipes from the site par file. This is performed by the local Scheme function `initialize-station-sites`, which is called by the Scheme language code:

```
(set! site-data-con (initialize-station-sites origin
station-list))
```

This function is a variant of the generic function `initialize-net-sites` described in “Initializing Sites” on page 46. It retrieves site information, reads travel-time tables, and initializes waveform containers. The returned container is used later to acquire waveforms and form the desired beams.

The function requires two arguments, which are described in Table 77. The input arguments are generic objects that contain the origin information and a list that contains the desired stations. The function returns an *initsite* container (symbol *site-data-con*). Each station has a corresponding *initsite* object, which collects the *dbsite*, *recipe*, *wfdata*, and *wfmem* objects for the station. These objects hold information about the waveforms and parameters for computing the beam for that station. Table 8 on page 50 lists the objects found in the *initsite* object for this application.

**TABLE 77: FUNCTIONAL INTERFACE OF INITIALIZE-STATION-SITES**

Argument Name	Usage	Type	Description
<i>origin</i>	input	GObj	<i>dborigin</i> object
<i>station-list</i>	input	list	station codes
n/a	returned value	container	<i>initsite</i> objects

## ▼ Seismic Processing

For more information about this function see “Detailed Description of Initialize Sites [3.2]” on page 170.

### Generate and Write Origin Beam [3.3]

This process performs the bulk of BOTF’s objectives. It generates and saves the beams via the local Scheme function `create-station-event-beam`. The function is called within the loop over stations (see Figure 20) by the Scheme code:

```
(create-station-event-beam origin site-data)
```

The function requires two arguments, which are described in Table 78. It writes the beams to the UNIX filesystem and inserts the corresponding records into the **wfdisc** database table.

**TABLE 78: FUNCTIONAL INTERFACE OF CREATE-STATION-EVENT-BEAM**

Argument Name	Usage	Type	Description
<i>origin</i>	input	GObj	<i>dborigin</i> object
<i>site-data</i>	input	GObj	<i>initsite</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

For more information about this function see “Detailed Description of Generate and Write Origin Beam [3.3]” on page 178.

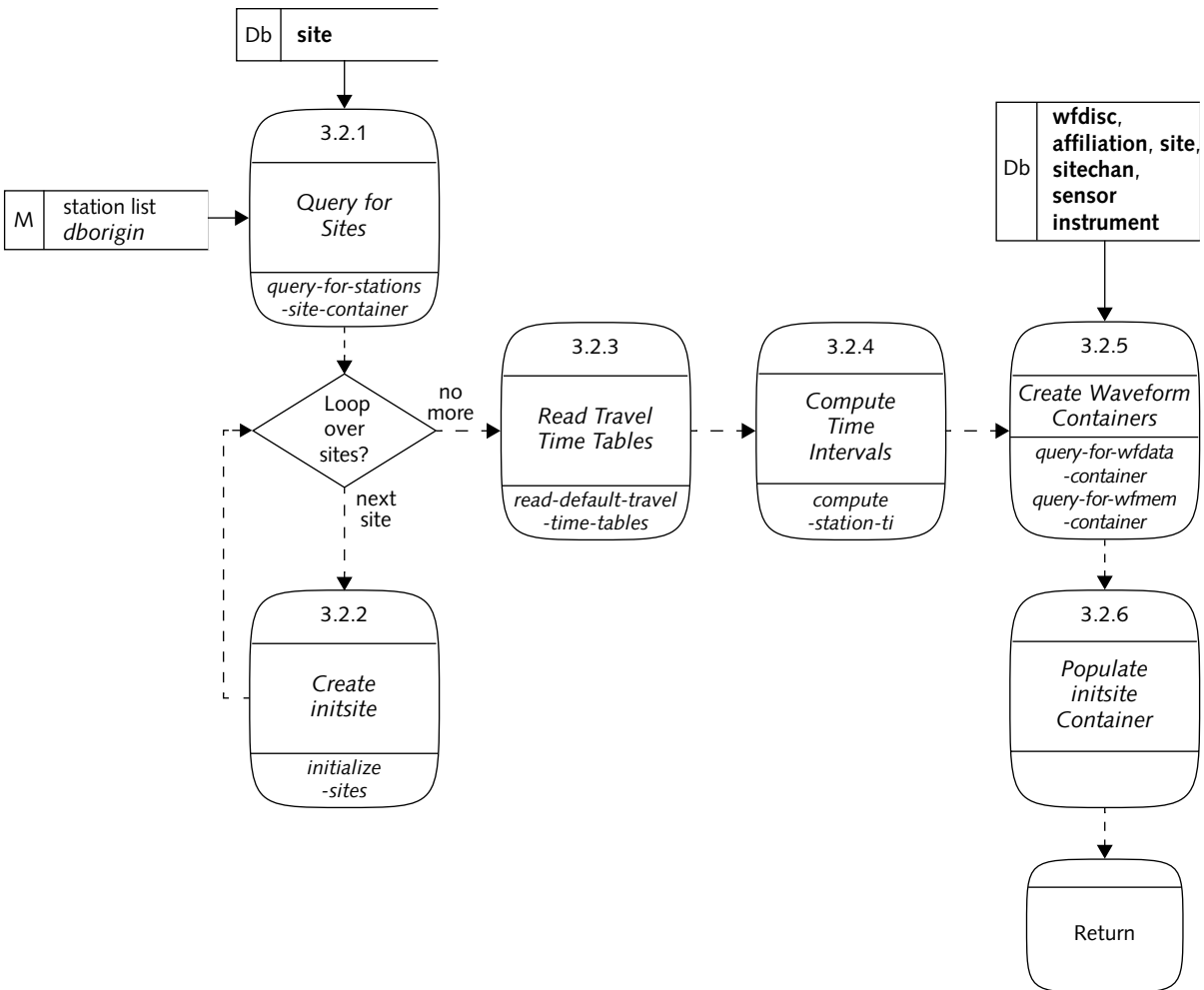
### Detailed Description of Initialize Sites [3.2]

This process retrieves site-specific information for all stations. It is implemented by the local Scheme function `initialize-station-sites`. Figure 21 shows the architecture of this process. The database is queried to find the site information corresponding to the input station list. The function then loops over the station list to retrieve site-specific recipes via the local Scheme function `initialize-sites`. After the recipe information has been obtained, the travel-time tables are read by

the *DFX/libutil* function `read-default-travel-time-tables`. The necessary data start-time and end-time are computed by the local Scheme function `compute-stations-ti`. The time intervals are then used to create and initialize the *wfdata* and *wfmem* containers. The channels to be excluded (specified by the user or in a par file) are removed from the *wfdata* and *wfmem* containers. An *initsite* container is formed where each object contains the relevant *wfdata* and *wfmem* objects.

The following subprocesses shown in Figure 21 are described in this section:

- *Query for Sites [3.2.1]*
- *Create initsite [3.2.2]*
- *Read Travel-time Tables [3.2.3]*
- *Compute Time Intervals [3.2.4]*
- *Create Waveform Containers [3.2.5]*
- *Populate initsite Container [3.2.6]*



### FIGURE 21. ARCHITECTURE OF INITIALIZE SITES



### Query for Sites [3.2.1]

This subprocess retrieves the records from the **site** table for the input stations. It creates a *dbsite* container via the local Scheme function `query-for-stations-site-container`. The function is called by the Scheme code:

```
(let* (stations-site-con
      (query-for-stations-site-container station-list
      origin-time)))
```

The function requires two arguments, which are described in Table 79. The query finds site records for stations in the station list whose ondate and offdate bound the origin time. The function returns a *dbsite* container.

**TABLE 79: FUNCTIONAL INTERFACE OF QUERY-FOR-STATIONS-SITE-CONTAINER**

Argument Name	Usage	Type	Description
<i>station-list</i>	input	list	station codes
<i>origin-time</i>	input	real	epoch time of origin
n/a	returned value	container	<i>dbsite</i> objects

For general information about how database queries are implemented see “Querying the Database” on page 41.

### Create initsite [3.2.2]

This subprocess creates the actual *initsite* object for each station. It is implemented via the local Scheme function `initialize-sites`, which is called from within a loop over stations. The *initsite* objects contain the site-specific recipes for checking the quality of and repairing the data, computing time intervals, and forming beams (see Table 8 on page 50). The function is called by the Scheme language code:

```
(insert-container initsite-con (initialize-sites origin
site))
```

## ▼ Seismic Processing

The function requires two arguments, which are described in Table 80. It takes the origin object and a site object for a specific station and then returns the *initsite* object.

**TABLE 80: FUNCTIONAL INTERFACE OF INITIALIZE-SITES**

Argument Name	Usage	Type	Description
<i>origin</i>	input	GObj	<i>dborigin</i> object
<i>site</i>	input	GObj	<i>dbsite</i> object
n/a	returned value	GObj	<i>initsite</i> object

**Read Travel-time Tables [3.2.3]**

This subprocess initializes the travel-time tables for later use. The travel-time tables are read by the *DFX/libutil* function `read-default-travel-time-tables` with input of the required phase identifiers supplied by the local Scheme function `get-phases-from-ti-and-beam-recs` operating on the *dbsite* container, symbol `stations-site-con`. These functions are invoked via the Scheme language call:

```
(read-default-travel-time-tables station-site-con
  (get-phases-from-ti-and-beam-recs initsite-com))
```

The functions require the three arguments listed in Table 81.

**TABLE 81: FUNCTIONAL INTERFACE OF READ-DEFAULT-TRAVEL-TIME-TABLES**

Argument Name	Usage	Type	Description
<i>station-site-con</i>	input	container	<i>dbsite</i> objects
(no explicit symbol)	input	list	phase names (strings) for which to read the travel-time tables
<i>initsite-con</i>	input	container	<i>initsite</i> objects
n/a	returned value	logical scalar	indicates success ( <code>t</code> ) or failure ( <code>nil</code> )

The second argument in Table 81 has no explicit symbol because it is supplied by the output from the function `get-phases-from-ti-and-beam-recs`, which returns a list of strings. The strings are the phase identifiers for which travel-time tables are to be read. These phase identifiers are extracted from the time-interval and beam recipes (their s- and e- phases) and beam recipes within the *initsite* container. For more information on reading travel-time tables see “Determining Data Time Intervals” on page 64.

### Compute Time Intervals [3.2.4]

Before creating the waveforms objects, the appropriate time window must be determined. This subprocess computes the minimum start-time and maximum end-time necessary to process all stations for the given origin via the local Scheme function `compute-stations-ti`. This function is called by the Scheme language code:

```
(set! stations-time-interval
  (compute-stations-ti initsite-con origin))
```

The function requires two arguments, which are described in Table 82. The *initsite* container holds the time-interval recipe and site locations. The function retrieves all of the seismic phases that may be used in later processing from the time-interval recipe. It predicts the arrival times of these phases from the origin and site locations. The minimum and maximum arrival times are returned as a list. These times will be used to create *waveform* containers in the next process.

**TABLE 82: FUNCTIONAL INTERFACE OF COMPUTE-STATIONS-TI**

Argument Name	Usage	Type	Description
<i>initsite-con</i>	input	container	<i>initsite</i> objects
<i>origin</i>	input	GObj	<i>dborigin</i> object
n/a	returned value	list	minimum and maximum arrival times

## ▼ Seismic Processing

**Create Waveform Containers [3.2.5]**

This subprocess creates two containers, *wfdata* and *wfmem*, for later use when acquiring waveforms. The *wfdata* container is created with the local Scheme function `query-for-wfdata-container`. A *wfdata* container collects the station/channel information from the **affiliation**, **instrument**, **sensor**, **site** and **sitechan** tables. This function is invoked via the Scheme language call:

```
(query-for-wfdata-container station-list chan-list t1 t2)
```

The function requires the four arguments listed in Table 83.

**TABLE 83: FUNCTIONAL INTERFACE OF QUERY-FOR-WFDATA-CONTAINER**

Argument Name	Usage	Type	Description
<i>station-list</i>	input	list	station codes
<i>chan-list</i>	input	list	channel codes
<i>t1</i>	input	real	start-time of the waveforms to be acquired
<i>t2</i>	input	real	end-time of the waveforms to be acquired
n/a	returned value	container	<i>wfdata</i> objects

This function returns a *wfdata* container with the information about each station/channel of the waveform data.

The local Scheme function `query-for-wfmem-container` creates the *wfmem* container. It is invoked via the Scheme language call:

```
(query-for-wfmem-container station-list chan-list t1 t2)
```

The function requires the four arguments in Table 84.

**TABLE 84: FUNCTIONAL INTERFACE OF QUERY-FOR-WFMEM-CONTAINER**

Argument Name	Usage	Type	Description
<i>station-list</i>	input	list	station codes
<i>chan-list</i>	input	list	channel codes
<i>t1</i>	input	real	start-time of the waveforms to be acquired
<i>t2</i>	input	real	end-time of the waveforms to be acquired
n/a	returned value	container	<i>wfmem</i> objects

This function returns a *wfmem* container. The *wfmem* objects hold the **wfdisc** records for each station and channel needed to form the requested beams. Once the waveforms are acquired, the *wfmem* objects reference the memory-resident time-series for each station/channel of waveforms.

This subprocess also removes any objects corresponding to channels in the parameter *excluded\_chans* from the two containers, *wfdata-con* and *wfmem-con*.

For more information see “Reading Waveforms” on page 58.

### **Populate initsite Container [3.2.6]**

This subprocess populates the *initsite* container with the *wfdata* and *wfmem* containers by looping over the *initsite* container for each site and inserting the appropriate *wfdata* and *wfmem* objects in each. This is accomplished in the same manner as described in “Network Initialization” on page 54.

### Detailed Description of Generate and Write Origin Beam [3.3]

The main objective of *BOTF* is accomplished within this process. Origin beams are generated by the local Scheme function `create-station-event-beam`. Figure 22 shows the architecture of this process. First, the necessary objects and parameters are extracted from the input *site* and *origin* containers for the current station. These include the *wfdata* and *wfmem* containers created during site initialization and the various recipe containers. Time intervals for processing are determined by the *DFX/libutil* function `compute-site-origin-ti` (3.3.1). The necessary waveforms are acquired and checked for quality if the quality-control flag is set (3.3.2). The actual beams are generated by the local Scheme function `compute-origin-beams` (3.3.4). Finally, the time series are written as binary files (with ".w" extensions) to the UNIX filesystem, and the corresponding *wfdisc* records are written to the output *wfdisc* table (3.3.5).

The following subprocesses shown in Figure 22 are described in this section:

- *Compute Time Intervals [3.3.1]*
- *Acquire Waveforms [3.3.2]*
- *Check Quality of Waveforms [3.3.3]*
- *Create Beam [3.3.4]*
- *Write to Database [3.3.5]*

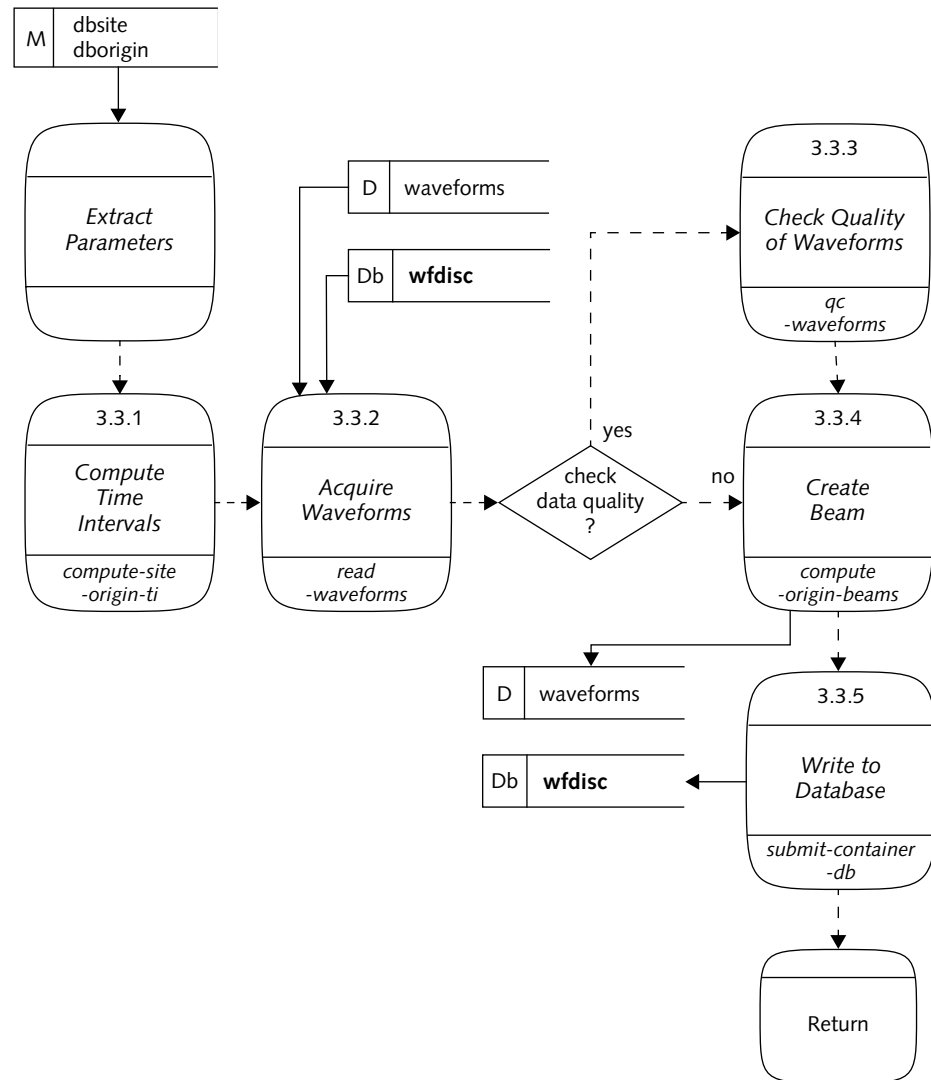


FIGURE 22. ARCHITECTURE OF GENERATE BEAM

## ▼ Seismic Processing

**Compute Time Intervals [3.3.1]**

This subprocess computes the time interval over which the beam is formed via a *DFX/libutil* function that is bound to the scheme symbol `compute-site-origin-ti`. The function is called by the Scheme language code:

```
(let*(time-interval (compute-site-origin-ti site origin  
tirec-con)))
```

The function requires three arguments. It takes the *site* and *origin* objects and the *time interval recipe* container as input and returns the minimum and maximum times that encompass the beam. The time interval recipe specifies the phase type and the lead and lag times to use in the computation. For more information about determining time intervals see “Determining Data Time Intervals” on page 64.

**Acquire Waveforms [3.3.2]**

This process acquires the waveforms to beamform. It is implemented by the *DFX/libdata* function `read-waveforms`, which is the standard *DFX* method (see “Reading Waveforms” on page 58). The *time interval* and the *wfdata* and *wfmem* containers are passed into the function. It places the waveforms in the *wfmem* objects and sets the *wfdata* objects to point at the appropriate waveform in the *wfmem* container.

**Check Quality of Waveforms [3.3.3]**

After the waveforms are acquired, their quality is checked, and if possible, they are repaired. To do this, the waveforms for stations that are listed in the parameter *qc-sta* are processed by the *DFX/libqc* function `qc-waveforms`. For a general description of this function see “Checking Quality of Waveforms” on page 69. The function is passed a *wfdata* container and the recipe for checking the quality of the data. The function `qc-waveforms` modifies the waveforms in the *wfdata* container by removing spikes and incorporating other data fixing algorithms.



**Create Beam [3.3.4]**

This subprocess accomplishes the objective of the application, namely beamforming the waveforms. Each beam is generated and written by the local Scheme function `compute-origin-beams`. The beams are computed in accordance with the time-interval recipes that provide the beam duration specifications and the beam recipes that provide the beam type, filter, and element specifications. The function uses the azimuth and slowness values determined from the *origin* and *site* containers to calculate the necessary time delays for time-aligning the filtered waveforms. See [IDC5.2.1] for more information on filtering and beamforming waveforms. The function is called by the Scheme language code:

```
(set! beam-output (compute-origin-beams wfdata-con site
                                     origin beamrec-con tirec-con))
```

The function requires five arguments, which are described in Table 85.

**TABLE 85: FUNCTIONAL INTERFACE OF COMPUTE-ORIGIN-BEAMS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>waveform</i> objects
<i>site</i>	input	GObj	<i>initsite</i> object
<i>origin</i>	input	GObj	<i>dborigin</i> object
<i>beamrec-con</i>	input	container	beam recipes
<i>tirec-con</i>	input	container	time-interval recipes
n/a	returned value	list	<i>dbwfdisc</i> and <i>dbw-ftag</i> objects

The current database connection is obtained by the common *libgdi* function `say-db-connection`. A *wfid* for each potential beam is obtained from the *lastid* table using the global *DFX* Scheme function `query-for-lastid`. Finally, the *DFX/libio* function `create-db-origin-beam` is called. This function computes and writes the beams to the UNIX filesystem and returns a container of **wfdisc** records that point to the new beams.

## ▼ Seismic Processing

**Write to Database [3.3.5]**

This subprocess saves the **wfdisc** records that point to the new beams. The records returned by the previous function are written to the **wfdisc** table by the global *DFX* Scheme function `submit-container-db`. This function is called by the Scheme language code:

```
(submit-container-db dbwfdisc-con out-wfdisc-table)
```

The function requires two arguments, which are described in Table 86.

**TABLE 86: FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (WFDISC)**

Argument Name	Usage	Type	Description
<i>dbwfdisc-con</i>	input	container	<i>dbwfdisc</i> objects
<i>out-wfdisc-table</i>	input	character-string	output table containing waveform file header and descriptive information generated by this application
n/a	returned value	logical scalar	indicates success ( <code>t</code> ) or failure ( <code>nil</code> )

The function takes as input the *dbwfdisc* container and the name of the table. The function returns either `t` or `nil` depending upon the success or failure of the submission. For more information about submitting results to the database see “Writing to the Database” on page 44.

## INTERACTIVE SEISMIC RECALL [4]

*Interactive Seismic Recall (IntSeisRcl)* is a DFX application that computes waveform features for arrivals added or modified by the analyst. The application source code resides in the file `DFX-int-recall.scm`. *IntSeisRcl* is typically called from within *ARS*. *ARS* sends an IPC message through the DACS containing the origin information (lat, lon, depth) and the names of the temporary tables that contain the arrival information. *IntSeisRcl* processes each arrival and updates their primary and secondary feature estimates in the temporary tables.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Initialize Travel-time Tables* [4.2]
- Detailed Description of *Perform Recall Signal Processing* [4.4]

### General Description

*IntSeisRcl* was designed to allow the analyst to update features for newly created or updated arrivals. The features are similar to those estimated in automatic seismic detection processing. The types of measurements are station dependent but can include arrival time uncertainty, best-beam calculations, f-k spectrum and polarization analysis, snr estimation, and various amplitude measures.

## ▼ Seismic Processing

**Architecture**

*IntSeisRcl* follows three general processes to estimate arrival features, as shown in Figure 23. First, input parameters are extracted (4.1), and the travel-time tables are initialized (4.2). The remaining processes occur within a double loop. The outer loop is over stations, and the inner loop is over arrivals for the current station. The site information and recipes are obtained within the outer loop (4.3). The main computational process is located within the double loop (4.4). This process, referred to as “recall” processing, acquires the waveforms, measures the new features, and writes the results to the temporary database tables.

**Input/Output**

A variety of parameters are needed by the process. The input CVAR datastore parameters used in the Scheme code are described in Table 87. These parameters are stored in the CVAR datastore and can be specified either on the command line or in “par” files. The table indicates the typical method used to specify each parameter.

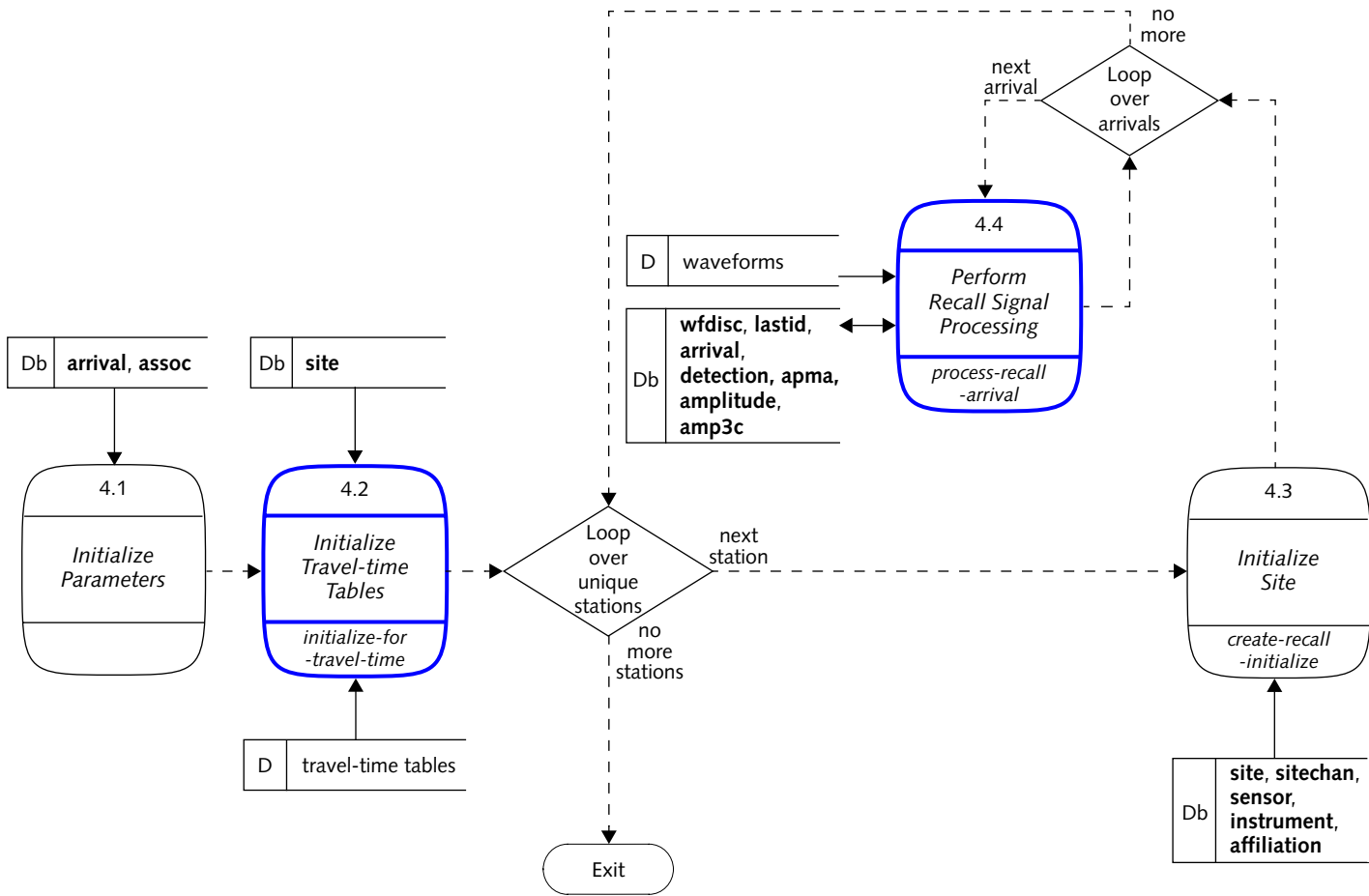


FIGURE 23. ARCHITECTURE OF INTSeisRCL

## ▼ Seismic Processing

**TABLE 87: INPUT PARAMETERS FOR INTERACTIVE SEISMIC RECALL PROCESSING**

Par Name	Source	Default Value	Description
<i>olat</i>	command line	<none>	origin latitude
<i>olon</i>	command line	<none>	origin longitude
<i>odepth</i>	command line	<none>	origin depth
<i>database-account</i>	command line	<none>	database access character-string
<i>out-arrival-table</i>	command line	<none>	<b>arrival</b> table
<i>out-detection-table</i>	command line	<none>	output table containing signal detection information generated by this application
<i>out-amplitude-table</i>	command line	<none>	output table containing amplitude information generated by this application
<i>out-apma-table</i>	command line	<none>	output table containing particle motion analysis information generated by this application
<i>out-amp3c-table</i>	command line	<none>	output table containing amplitude estimates (from 3-C stations) generated by this application
<i>assoc-table</i>	command line	<none>	input association table
<i>affiliation-table</i>	par file	<none>	table containing station and network affiliations
<i>instrument-table</i>	par file	<none>	table containing generic (default) calibration information about a station
<i>sensor-table</i>	par file	<none>	table containing specific calibration information for physical channels

**TABLE 87: INPUT PARAMETERS FOR INTERACTIVE SEISMIC RECALL PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>site-table</i>	par file	<none>	table containing site-location information
<i>sitechan-table</i>	par file	<none>	table containing station-channel information
<i>in-wfdisc-table</i>	par file	<none>	input table containing waveform file header and descriptive informations
<i>wfdisc-extension-len</i>	par file	86400	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>recall-auth</i>	par file	<none>	string to enter into <i>auth</i> field in output arrival table
<i>NetType</i>	par file	<none>	network type, as in: "ss", "array", or "larray"
<i>StaType</i>	par file	<none>	station type, as in: "1c", "3c"
<i>WaveType</i>	par file	" "	wave type (" ", "hydro-", or "infra-")
<i>recall-recompute-deltim</i>	par file	0	flag used to recompute <i>deltim</i>
<i>depth-phase-snr-amprec-list</i>	par file	<none>	amplitude recipe for computing depth-phase snr
<i>depth-phase-snr-amptype</i>	par file	db-snr	amplitude type to use in computing depth-phase snr
<i>recall-site-recipe-file</i>	par file	<none>	par file pointing to site-specific recipe files

## ▼ Seismic Processing

**TABLE 87: INPUT PARAMETERS FOR INTERACTIVE SEISMIC RECALL PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>recall-depth-phases</i>	par file	pP, sP, pPKP, pPKPbc, pPKiKP	valid depth-phase identifiers
<i>recall-detection-beams</i>	par file	<none>	detection beam recipes

The results of the processing are written to the database. Table 88 lists the database tables that may be updated; however, the specific tables that are actually updated depend on the station type and the input parameters. Table 233 on page 446 lists the specific attributes written or updated for each of these database tables.

**TABLE 88: DATA WRITTEN BY INTERACTIVE SEISMIC RECALL PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>amplitude</b>	database table	waveform amplitude for each detection
<b>amp3c</b>	database table	waveform amplitude for each detection (on 3-C stations)
<b>apma</b>	database table	particle motion analysis attributes of each detection
<b>arrival</b>	database table	arrival attributes of each detection
<b>detection</b>	database table	detection attributes of each detection



## Processes

The following processes shown in Figure 23 on page 185 are described in this section:

- *Initialize Parameters [4.1]*
- *Initialize Travel-time Tables [4.2]*
- *Initialize Site [4.3]*
- *Perform Recall Signal Processing [4.4]*

### Initialize Parameters [4.1]

This process initializes various internal variables, extracts user parameters from the CVAR datastore, and retrieves arrival information from the database. Input parameters are extracted and variables are initialized by inline Scheme code. A *dborigin* object is created using the command line parameters specified at run-time

The arrival information is extracted from the database and placed into *dbarrival* and *dbassoc* containers by two local Scheme functions: *query-for-dbarrival-container* and *query-for-dbassoc-container*. These functions have no input parameters. Instead they obtain the relevant table names directly from the CVAR datastore. The tables are assumed to be temporary tables that only contain the arrivals of interest, and therefore *IntSeisRcl* retrieves the entire **arrival** and **assoc** tables. The functions *query-for-dbarrival-container* and *query-for-dbassoc-container* return a *dbarrival* and a *dbassoc* container, respectively. For general information on how database queries are implemented see “Database Operations” on page 40.

### Initialize Travel-time Tables [4.2]

This process initializes the travel-time tables so azimuth and slowness values can be predicted in later processing. It creates a *dbsite* container and initializes the travel-time table information for all relevant sites and seismic phases. This process is implemented by the local Scheme function *initialize-for-travel-time*. The function is called by the Scheme language code:

## ▼ Seismic Processing

```
(set! dbsite-con (initialize-for-travel-time
  dbarrival-con))
```

The predicted azimuth and slowness information is used later to calculate the highest SNR beam. The function requires one argument, which is described in Table 89. The *dbarrival* container is the one obtained in the previous step, *Initialize Parameters [4.1]*. The function returns a *dbsite* container that includes all stations in the *arrival* container. In addition, the travel-time tables are initialized for later use.

**TABLE 89: FUNCTIONAL INTERFACE OF INITIALIZE-FOR-TRAVEL-TIME**

Argument Name	Usage	Type	Description
<i>dbarrival-con</i>	input	container	<i>dbarrival</i> objects
n/a	returned value	container	<i>dbsite</i> objects

For more information on this function see “Detailed Description of Initialize Travel-time Tables [4.2]” on page 192.

### Initialize Site [4.3]

This process retrieves site-specific information from the database and recipes from the CVAR datastore. It is implemented by the local Scheme function `create-recall-initsite`. The function is called by the Scheme language code:

```
(set! initsite (create-recall-initsite sta
  sta-dbarrival-con sta-dbsite dborigin))
```

The function reads a variety of site-specific recipes that are used later to get waveforms, compute beams, perform f-k analysis, and perform other computations. The recipes are inserted into the *initsite* object and are read using the file specified by the CVAR datastore parameter *recall-site-recipe-file*. The function is in the outer loop in Figure 23 on page 185 and therefore is called for each station. The fourth column of Table 8 on page 50 delineates those entities contained in the *initsite* objects created by this function for *IntSeisRcl*. For more information about this type of function see “Initializing Sites” on page 46.

The function `create-recall-initsite` has four arguments, which are described in Table 90. The function is called for a single station (the current station from the outer loop). The arrivals for this station have been extracted and placed in their own container (symbol *sta-dbarrival-con*). Similarly, the site information for this station is placed in symbol *sta-dbsite*, which is passed into the Scheme function. The *origin* object is the one created in the initialization step.

**TABLE 90: FUNCTIONAL INTERFACE OF CREATE-RECALL-INITSITE**

Argument Name	Usage	Type	Description
<i>sta</i>	input	character-string	station code
<i>sta-dbarrival-con</i>	input	container	<i>dbarrival</i> objects for the current station
<i>sta-dbsite</i>	input	GObj	<i>dbsite</i> object for current station
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
n/a	returned value	GObj	<i>initsite</i> object

The Scheme function returns an *initsite* object. The *initsite* object holds all of the information needed to make the various estimates for this station. This includes station information as well as various site-specific recipes. It also contains a *wfdata* container that will be used for acquiring waveforms. Table 8 on page 50 lists the specific objects in this application's *initsite* object.

#### Perform Recall Signal Processing [4.4]

This process accomplishes the main objective of the application. It computes the feature measurements for each arrival via the local Scheme function `process-recall-arrival`. It is called for each arrival belonging to the current station. The function's tasks include acquiring the waveforms, finding the highest SNR beam, estimating *deltim*, performing f-k or polarization analysis, and making amplitude measurements. The results are written to the appropriate tables in the database. The function is called by the Scheme language code:

## ▼ Seismic Processing

```
(process-recall-arrival initsite dbarrival dbassoc-con)
```

The function requires three arguments, which are described in Table 91. The *initsite* object was obtained in the previous process, *Initialize Site* [4.3]. The *dbarrival* object is extracted from the arrivals container for this station. The container holding all of the associations (symbol *dbassoc-con*) is passed directly to the function.

**TABLE 91: FUNCTIONAL INTERFACE OF PROCESS-RECALL-ARRIVAL**

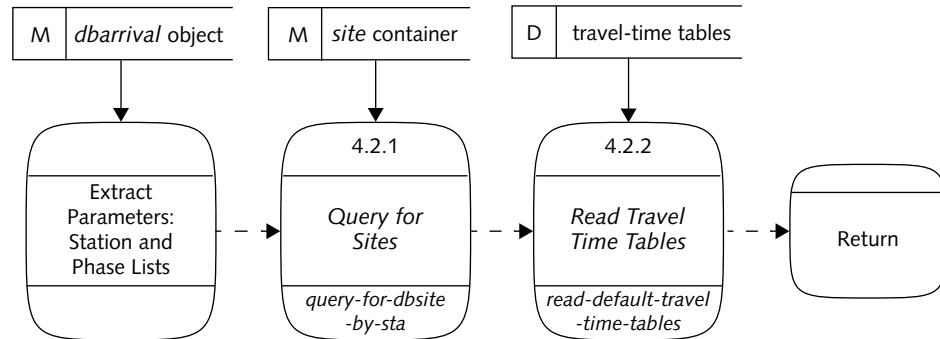
Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>dbarrival</i>	input	GObj	<i>dbarrival</i> object
<i>dbassoc-con</i>	input	container	<i>dbassoc</i> objects
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The function writes the various estimates to the temporary database tables specified on input. It returns t upon successful completion of the function.

For more information about this function see “Detailed Description of Perform Recall Signal Processing [4.4]” on page 194.

### Detailed Description of Initialize Travel-time Tables [4.2]

This process reads the travel-time tables into memory so they can be used to predict azimuth and slowness values in later processing. Figure 24 shows the architecture of the process. Travel-time tables are initialized in three sequential subprocesses. The stations and phases are extracted from the input *dbarrival* container. The database is queried to obtain *site* objects for each station contained in the symbol *sta-list*. Then the travel-time tables are initialized for these stations and phases. The *dbsite* container is returned to the calling function.



**FIGURE 24. ARCHITECTURE OF INITIALIZE TRAVEL TIMES**

The following subprocesses shown in Figure 24 are described in this section:

- *Query for Sites [4.2.1]*
- *Read Travel-time Tables [4.2.2]*

#### **Query for Sites [4.2.1]**

This subprocess retrieves the records from the **site** table for the input stations. The *dbsite* container is created by the local Scheme function `query-for-dbsite-by-sta`. This function queries the table specified in the CVAR parameter *site-table* and places the results into a *dbsite* container. The query is only restricted by the station list (symbol *sta-list*). For general information on database queries in *DFX* see “Database Operations” on page 40.

#### **Read Travel-time Tables [4.2.2]**

This subprocess initializes the travel-time tables for later use. The travel-time tables are read by the *DFX/libutil* function `read-default-travel-time-tables`. The input parameters to this function are the *dbsite* container from the previous step and the phase identifiers extracted from the *arrival* container. The tables are saved in memory and can be accessed later by other *DFX* functions. For general information on reading travel-time tables see “Determining Data Time Intervals” on page 64.

### Detailed Description of Perform Recall Signal Processing [4.4]

This process performs the arrival feature estimation. Figure 25 shows the architecture of this process. Five subprocesses are performed sequentially. The actual measurements that are computed are station and phase dependent. The first subprocess determines if the arrival is a depth phase. Next, the waveforms are acquired. Then several beams are computed, which will be used to make most of the subsequent measurements. The beams include the “best” beam (defined as the beam with the highest snr) and the standard beams that are defined in the beam recipes. The final two subprocesses estimate the various features and write the results to the database.

The following subprocesses shown in Figure 25 are described in this section:

- *Identify Depth-phase Arrivals [4.4.1]*
- *Acquire Waveforms [4.4.2]*
- *Identify Best Beam [4.4.3]*
- *Revise Detection Attributes [4.4.4]*
- *Estimate Arrival Features and Write to Database [4.4.5]*

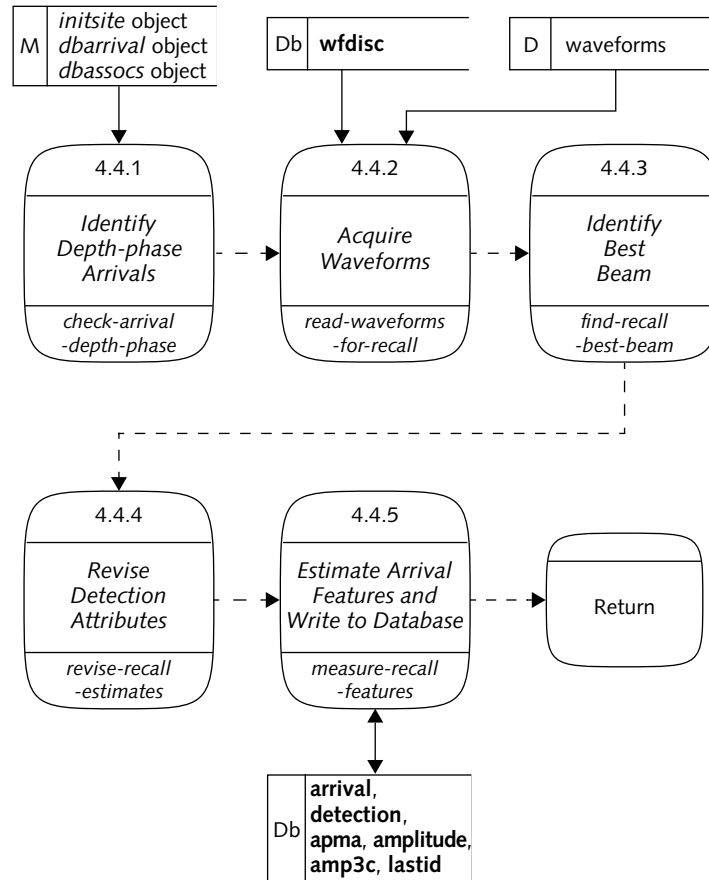


FIGURE 25. ARCHITECTURE OF PERFORM RECALL SIGNAL PROCESSING

**Identify Depth-phase Arrivals [4.4.1]**

This subprocess identifies depth phases and creates a flag, *is-a-depth-phase*, to mark these arrivals as needing additional processing. Depth-phase arrivals are identified by the local Scheme function *check-arrival-depth-phase*. The function checks if the *phase* attribute from the **assoc** table matches one of the phase identifiers in the CVAR parameter *recall-depth-phases*. The flag *is-a-depth-phase* is used later to determine if the depth-phase snr should be calculated.

## ▼ Seismic Processing

**Acquire Waveforms [4.4.2]**

This subprocess acquires the appropriate waveforms that will be used to make the feature measurements. It is implemented by a local Scheme function `read-waveforms-for-recall`, which acquires waveforms by the standard method in *DFX* (see “Reading Waveforms” on page 58). The function determines an appropriate time interval by using the detection recipe and the arrival time. It creates a *wfmem* container via the global Scheme function `query-for-wfmem-by-sta-chan-time`. The waveforms are read by the *DFX/libdata* function `read-waveforms`. This function is called by the Scheme language code:

```
(set! wfdata-con (read-waveforms-for-recall initsite
dbarrival))
```

This function calls the function `read-waveforms`, which is passed the time interval and the *wfdata* and *wfmem* containers. It places the waveforms in the *wfmem* container and sets the *wfdata* container to point at the appropriate waveform in *wfmem*. The quality of the waveforms is checked, and if possible, repaired by the *DFX/libqc* function `qc-waveforms`. The function returns a *wfdata* container.

The function `read-waveforms-for-recall` requires two arguments, which are described in Table 92.

**TABLE 92: FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-RECALL**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>dbarrival</i>	input	GObj	<i>dbarrival</i> object
n/a	returned value	container	<i>waveforms</i> objects

**Identify Best Beam [4.4.3]**

This subprocess finds the “best” detection beam via the local Scheme function `find-recall-best-beam`. This detection beam is used later for feature measurements. The function first computes the standard beams that are specified in



the *beam recipe* container. It then loops over each recipe in the *detection-beam* container (detection and standard beams are separate entities). The azimuth and slowness from the arrival table are used to steer the detection beams unless they are null, in which case the predicted azimuth and slowness from the origin is used. The beam recipe whose beam yields the highest snr is saved in the *detection* object as the best beam. The function is called by the Scheme language code:

```
(find-recall-best-beam wfdata-con initsite dbarrival det)
```

The function requires four arguments, which are described in Table 93. The *detection* object is created from the *dbarrival* object before calling the function. All of the other objects are from earlier processing. Both the *wfdata* container and *initsite* object are modified within the function. The standard beams are placed in the *wfdata* container, and the best-beam recipe is placed in the *detection* object along with the snr.

**TABLE 93: FUNCTIONAL INTERFACE OF FIND-RECALL-BEST-BEAM**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input and output	container	<i>waveform</i> objects
<i>initsite</i>	input and output	GObj	<i>initsite</i> object
<i>dbarrival</i>	input	GObj	<i>dbarrival</i> object
<i>det</i>	input and output	GObj	<i>detection</i> object
n/a	returned value	GObj	<i>detection</i> object (null value indicates failure)

This function returns the *detection* object, which was originally input, with the best beam and snr attributes set, or *nil* on error.

## ▼ Seismic Processing

**Revise Detection Attributes [4.4.4]**

This subprocess computes the primary arrival features via the local Scheme function `revise-recall-estimates`. These features include: (i) uncertainty in arrival time, (ii) amplitude for the “best” beam and the f-k beam, (iii) azimuth and its uncertainty, and (iv) slowness and its uncertainty. The function is called by the Scheme language code:

```
(revise-recall-estimates wfdata-con initsite det)
```

This function requires three arguments, which are described in Table 94. These arguments are used for both input and output.

**TABLE 94: FUNCTIONAL INTERFACE OF REVISE-RECALL-FEATURES**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input and output	container	<i>waveform</i> objects
<i>initsite</i>	input and output	GObj	<i>initsite</i> object
<i>det</i>	input and output	GObj	<i>detection</i> object
n/a	returned value	GObj	<i>detection</i> object (null value indicates error)

To compute the features, `revise-recall-features` calls several functions. These functions are listed in Table 95.

**TABLE 95: FUNCTIONS CALLED WITHIN REVISE-RECALL-FEATURES**

Function	Type
<code>calc-delttime</code>	<i>DFX/libdetect</i>
<code>compute-det-amp-best-beam</code>	global Scheme
<code>make-fk</code>	<i>DFX/libfk</i>
<code>make-beamform-fk</code>	<i>DFX/libfk</i>

**TABLE 95: FUNCTIONS CALLED WITHIN REVISE-RECALL-  
FEATURES (CONTINUED)**

Function	Type
<code>compute-det-amp-fk-beam</code>	global Scheme
<code>measure-detection-polarization</code>	<i>DFX/libthreec</i>

The *DFX/libdetect* function `calc-deltim` estimates the arrival time uncertainty. It is only executed if the CVAR parameter *recompute-deltim?* is set to true (t). The attribute *deltim* in the *detection* object is updated.

Next, the global *DFX* function `compute-det-amp-best-beam` determines the amplitude of the best beam for all arrivals and stores it in the *detection* object. See [IDC5.2.1] for more information about estimating waveform amplitude.

An f-k analysis is performed for analyst-added arrivals at array stations if it was not computed earlier. The analysis determines the beam with the highest F-statistic (within the constraints of the f-k recipe) and stores it in the *detection* object. The amplitude of the f-k beam is determined for all arrivals from array stations whether or not they were added by the analyst. See [IDC5.2.1] for more information about generating frequency-wavenumber spectra from arrays of seismic stations.

Polarization analysis is performed for arrivals at 3-C seismic stations. The polarization estimate for an single 3-C seismic station is determined from the covariance matrix between the three components. The polarization for an array is an average of this covariance matrix over the individual elements of the array. If the analyst has made azimuth and slowness measurements on a single 3-C station, the analyst's results are used instead of *DFX*'s results for the detection. See [IDC5.2.1] for more information on performing polarization analysis from 3-C seismic stations.

## ▼ Seismic Processing

**Estimate Arrival Features and  
Write to Database [4.4.5]**

This subprocess creates and fills database objects, which are then written to the appropriate database tables. It is implemented by the local Scheme function `measure-recall-features`. The subprocess first initializes the database objects and then fills the appropriate objects for the station and network type via several functions (see Table 96). Detection amplitude measures are made for the arrival and depth-phase snr measurements (if appropriate). Finally the results are written to the database tables.

**TABLE 96: DATABASE OBJECTS DEFINED IN MEASURE-RECALL-FEATURES**

Database Object/Container	Populated by
<i>dbarrival</i>	DFX/libdb function create-dbarrival-from-detection
<i>dbdetection</i>	DFX/libdb function create-dbdetection-from-detection
<i>dbapma</i>	DFX/libdb function create-dbapma-from-detection
<i>dbamplitude-con</i>	global Scheme function make-det-amplitudes
<i>depth-phase-dbamplitude-con</i>	local Scheme function make-depth-phase-amplitude
<i>dbamp3c-con</i>	DFX/libdb function create-dbamp3c-from-detection
<i>sbsnr-con</i>	global Scheme function create-dbsnr-dbamplitude-from-det

The function is called by the Scheme language code:

```
(measure-recall-features wfdata-con initsite det
  is-a-depth-phase)
```

The function `measure-recall-features` requires four arguments, which are described in Table 97. The function inserts the new objects into the previously opened database. The database tables that are written to may include **arrival**, **detection**, **apma**, **amplitude**, and **amp3c**. The exact tables depend on the station and phase types. For more information about submitting results to the database see “Writing to the Database” on page 44.

**TABLE 97: FUNCTIONAL INTERFACE OF MEASURE-RECALL-FEATURES**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>waveform</i> objects
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>det</i>	input	GObj	<i>detection</i> object
<i>is-a-depth-phase</i>	input	number	flag indicating depth phase
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

## AUTOMATIC SEISMIC RECALL [5]

*Automatic Seismic Recall (AutoSeisRcl)* is a DFX application that computes waveform feature measurements after the analyst review. The application's source code resides in the file `DFX-recall.scm`. It is usually executed in an automated pipeline that is queued after analyst review to compute features for analyst-added arrivals. *AutoSeisRcl* retrieves arrivals and finds the associated origin and site information. For each arrival/station pair, the "best" beam is found, the primary attributes are estimated and screened, and secondary features are estimated. The results are then written to the database.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Perform Recall Signal Processing* [5.4]

### General Description

*AutoSeisRcl* was designed to estimate features for arrivals modified or added by the analyst. The features are similar to those estimated in automatic seismic detection processing and interactive seismic recall processing. The actual estimates made are station dependent but can include: (i) arrival time uncertainty, (ii) best-beam calculations, (iii) f-k and polarization analysis, and (iv) snr and various amplitude measures.

## Architecture

Figure 26 shows the architecture of *AutoSeisRcl*. The general flow consists of an initialization step followed by a double loop over stations and arrivals where the recall processing actually occurs. The initialization step consists of three primary processes: the database is queried for the appropriate arrivals (5.1); a process is executed to set up azimuth and slowness residual updating if the user has set the input flag for residual updates (5.2); and a station list is extracted from the arrivals that were obtained by the previous query.

The main automatic seismic recall data processing occurs inside the double loop. The outer loop is a mapping over stations obtained in the initialization step. In this loop, associations from the `assoc` table that belong to the current station are extracted along with the corresponding origins. Next, an *initsite* object is created for the current station (5.3). The main processing function `process-recall-arrival` is executed within the inner loop, which is a mapping over arrivals belonging to the current station (5.4).

## Input/Output

Table 98 lists the input CVAR datastore parameters used in this application. *AutoSeisRcl* processes a set of arrivals. Three methods can be used to specify the set of arrivals. The parameters used to define the arrivals are generally specified on the command line. The remaining parameters are set or referenced in the `DFX-recall.par` file. *AutoSeisRcl* loads the `DFX-site-detection.par` file, which in turn loads in the necessary recipes (not listed here).

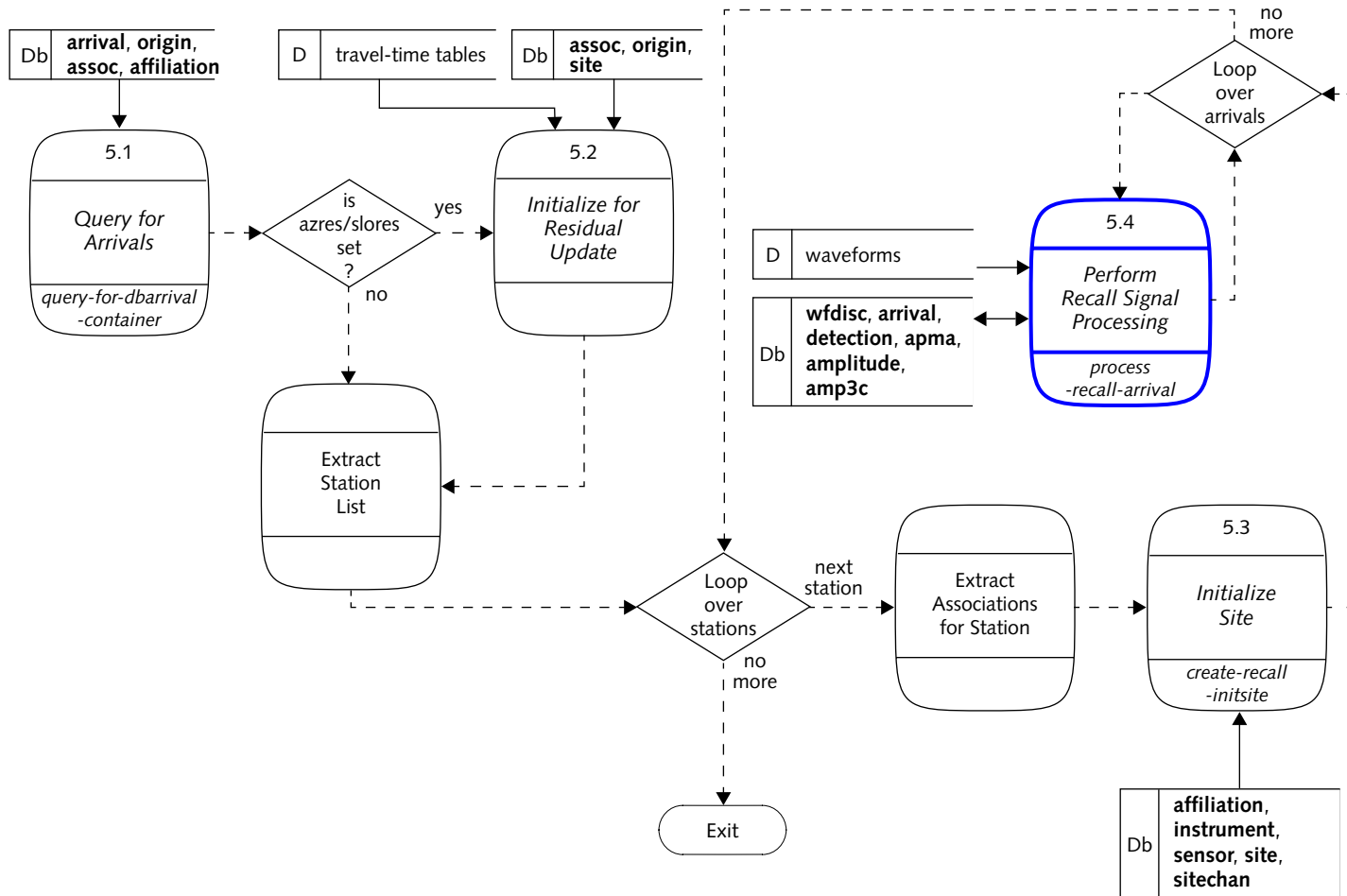


FIGURE 26. ARCHITECTURE OF AUTOSeisRCL



**TABLE 98: INPUT PARAMETERS FOR AUTOMATIC SEISMIC RECALL PROCESSING**

Par Name	Source	Default Value	Description
<i>start-time</i>	command line	<none> <sup>1</sup>	start-time of the processing interval
<i>end-time</i>	command line	<none> <sup>1</sup>	end-time of the processing interval
<i>net</i>	command line	<none> <sup>1</sup>	network of stations to process
<i>arid</i>	command line	<none> <sup>1</sup>	unique arrival identification number
<i>arrival-query</i>	command line	<none> <sup>1</sup>	database query to retrieve desired arrivals
<i>database-account</i>	par file	<none>	database access character-string
<i>affiliation-table</i>	par file	<b>affiliation</b>	table containing station and network affiliations
<i>instrument-table</i>	par file	<b>instrument</b>	table containing generic (default) calibration information about a station
<i>sensor-table</i>	par file	<b>sensor</b>	table containing specific calibration information for physical channels
<i>site-table</i>	par file	<b>site</b>	table containing site-location information
<i>sitechan-table</i>	par file	<b>sitechan</b>	table containing station-channel information
<i>origin-table</i>	par file	<b>origin</b>	input <b>origin</b> table
<i>lastid-table</i>	par file	<b>lastid</b>	<b>lastid</b> table for getting unique identification numbers
<i>in-arrival-table</i>	par file	<b>arrival</b>	input <b>arrival</b> table
<i>in-assoc-table</i>	par file	<b>assoc</b>	input association table

## ▼ Seismic Processing

**TABLE 98: INPUT PARAMETERS FOR AUTOMATIC SEISMIC RECALL PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information
<i>out-amplitude-table</i>	par file	amplitude	output table containing amplitude information generated by this application
<i>out-amp3c-table</i>	par file	amp3c	output table containing amplitude estimates (from 3-C stations) generated by this application
<i>out-apma-table</i>	par file	apma	output table containing particle motion analysis information generated by this application
<i>out-arrival-table</i>	par file	arrival	output <b>arrival</b> table
<i>out-detection-table</i>	par file	detection	output table containing signal detection information generated by this application
<i>recall-auth</i>	par file	<none>	character-string to enter into <i>auth</i> attribute in output <b>arrival</b> table
<i>wfdisc-extension-len</i>	par file	86400	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>NetType</i>	par file	<none>	network type, as in: "ss", "array", or "larray"
<i>StaType</i>	par file	<none>	station type, as in: "1c", "3c"

**TABLE 98: INPUT PARAMETERS FOR AUTOMATIC SEISMIC RECALL PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>WaveType</i>	par file	" "	wave type (" ", "hydro-", "infra-")
<i>recall-recompute-deltim</i>	par file	0	flag used to recompute <i>deltim</i>
<i>recall-set-azres-slores</i>	par file	1	flag used to set azimuth and slowness residuals
<i>recall-use-snr-thresh</i> <i>-best-beam</i>	par file	0	flag used to apply the snr threshold from the best beam
<i>recall-site-recipe-file</i>	par file	<none>	par file pointing to site specific recipe files

1. The arrivals to be processed must be specified in one of three ways. In order of precedence they are: (*arid*), (*arrival-query*), or (*start-time*, *end-time*, *net*). The IDC uses (*start-time*, *end-time*, *net*).

The results of the processing are written to the database. Table 99 lists the database tables that may be updated; however, the specific tables that are actually updated depend on the station type and the input parameters. Table 234 on page 447 indicates the specific attributes written or updated for each of these database tables.

**TABLE 99: DATA UPDATED BY AUTOMATIC SEISMIC RECALL PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>amplitude</b>	database table	waveform amplitude for each detection
<b>amp3c</b>	database table	waveform amplitude for each detection (on 3-C stations)

## ▼ Seismic Processing

**TABLE 99: DATA UPDATED BY AUTOMATIC SEISMIC RECALL PROCESSING (CONTINUED)**

Name	Category	Description
<b>apma</b>	database table	particle motion analysis attributes of each detection
<b>arrival</b>	database table	arrival attributes of each detection
<b>detection</b>	database table	detection attributes of each detection

**Processes**

The following processes shown in Figure 27 on page 214 are described in this section:

- *Query for Arrivals [5.1]*
- *Initialize for Residual Update [5.2]*
- *Initialize Site [5.3]*
- *Perform Recall Signal Processing [5.4]*

**Query for Arrivals [5.1]**

This process retrieves the records from the **arrival** table that are to be processed. It is implemented by a call to the local Scheme function `query-for-dbarrival-container`. The process constructs a suitable database query based on the input parameters set in the CVAR datastore to obtain the appropriate arrival information. See “Querying the Database” on page 41 for more information about database objects and queries. The function is called by the Scheme language code:

```
(set! dbarrival-con (query-for-dbarrival-container))
```

The function requires no arguments, and a *dbarrival* container is returned.

The function `query-for-dbarrival-container` acquires the nine parameters described in Table 100 via access to a CVAR datastore. Three possible queries can be formed based on the parameters that are set in the CVAR datastore. They are listed in order of precedence in Table 101.

The function applies the query to the database via the global Scheme function `query-for-container`. It returns a *dbarrival* container with information from the *arrival* table.

**TABLE 100: INTERNAL SYMBOLS FOR QUERY-FOR-DBARRIVAL-CONTAINER**

Symbol	Type	Description
<i>start-time</i>	real	start-time of this processing interval
<i>end-time</i>	real	end-time of the processing interval
<i>origin-table</i>	character-string	name of the <b>origin</b> table
<i>assoc-table</i>	character-string	name of the association table
<i>arrival-table</i>	character-string	name of the <b>arrival</b> table
<i>affiliation-table</i>	character-string	table containing station and network affiliations
<i>arid</i>	integer	unique arrival identification number
<i>net</i>	character-string	name of the network
<i>arrival-query</i>	character-string	user defined query

**TABLE 101: POSSIBLE ARRIVAL QUERIES**

Parameters Set in CVAR	Query Description
<i>arid</i>	query for arrival with the given <i>arid</i>
<i>arrival-query</i>	query based on user provided query
<i>start-time, end-time, net</i>	query for arrivals that were added by the analyst (in ARS) at stations in the network, <i>net</i> , and are associated to origins that occurred between <i>start-time</i> and <i>end-time</i> ; excludes arrivals that were previously processed by seismic recall processing or arrivals that are long-period phases like 'LR'

## ▼ Seismic Processing

**Initialize for Residual Update [5.2]**

This process initializes the objects and travel-time tables needed to calculate arrival time, azimuth, and slowness residuals. The residuals are the difference between the observations and the predictions. The predictions are based on the origin and station locations. The process is implemented by the local Scheme function `initialize-for-residual-update`. This function is executed if the CVAR parameter `recall-set-azres-slores` is set to a positive value or is not set at all. If it is set to a zero or a negative value then the function is not executed. The function is called by the Scheme language code:

```
(residual-output-list (initialize-for-residual-update
  dbarrival-con))
```

The function takes the single argument described in Table 102 and returns a list having three containers: a *dbassoc* container, a *dborigin* container, and a *dbsite* container.

**TABLE 102: FUNCTIONAL INTERFACE OF INITIALIZE-FOR-RESIDUAL-UPDATE**

Argument Name	Usage	Type	Description
<i>dbarrival-con</i>	input	container	<i>dbarrival</i> objects
n/a	returned value	list	list of <i>dbassoc</i> , <i>dborigin</i> , and <i>dbsite</i> containers

The function retrieves associations from the **assoc** table in the database that correspond to the arrivals in the input *dbarrival* container. Only associations that have residual attributes set to *NULL* (*timeres*, *azres*, and *slores*) are placed in a *dbassoc* container. The origin records corresponding to the *dbassoc* objects are retrieved from the **origin** table. The function retrieves the records from the **site** table for all stations in the *dbassoc* container. It then initializes the travel-time tables for those stations. The local Scheme functions that perform the database queries are listed in Table 103. See “Querying the Database” on page 41 for more information about

database queries. The travel-time tables are initialized by the *DFX/libutil* function `read-default-travel-time-tables`. See “Determining Data Time Intervals” on page 64 for information about this function.

**TABLE 103: DATABASE QUERY FUNCTIONS USED BY INITIALIZE-FOR-RESIDUAL-UPDATE**

Scheme Function	Input	Container Types Returned By Query
<code>query-for-dbassoc-with-null-residual-by-arids</code>	<code>arid-list</code>	<code>dbassoc-con</code>
<code>query-for-dborigin-by-orid</code>	<code>orid-list</code>	<code>dborigin-con</code>
<code>query-for-dbsite-by-sta</code>	<code>sta-list</code>	<code>dbsite-con</code>

### Initialize Site [5.3]

Initializing site information is a standard procedure in most *DFX* Scheme applications. The purpose is to generate an *initsite* object that contains site-specific information for later processing. Table 8 on page 50 lists the entities contained in the *initsite* object. For further information on *initsite* objects see “Initializing Sites” on page 46.

In *AutoSeisRcl*, this process is implemented by the local Scheme function `create-recall-initsite`. It is called using the Scheme language code:

```
(set! initsite (create-recall-initsite sta
  sta-dbarrival-con sta-dbsite sta-dbassoc-con
  sta-dborigin-con))
```

This function requires five arguments, which are described in Table 104, and returns an *initsite* object.

## ▼ Seismic Processing

TABLE 104: FUNCTIONAL INTERFACE OF CREATE-RECALL-INITSITE

Argument Name	Usage	Type	Description
<i>sta</i>	input	character-string	station code
<i>sta-dbarrival-con</i>	input	container	<i>dbarrival</i> objects for the input station
<i>sta-dbsite</i>	input	GObj	<i>dbsite</i> object
<i>sta-dbassoc-con</i>	input	container	<i>dbassoc</i> objects
<i>sta-dborigin-con</i>	input	container	<i>dborigin</i> objects
n/a	returned value	GObj	<i>initsite</i> object

**Perform Recall Signal Processing [5.4]**

This process performs the actual feature estimation via the local Scheme function `process-recall-arrival`. It is called for each arrival belonging to the current station. The function's tasks include acquiring the waveforms, finding the highest snr beam, estimating the arrival time uncertainty, performing f-k or polarization analysis, and making amplitude measurements. The results are written to the appropriate table in the database.

The function is called from within the inner loop of the main application by the Scheme language code:

```
(process-recall-arrival initsite dbarrival)
```

The function requires two arguments, which are described in Table 105. The *initsite* object obtained in the previous process provides the site-specific and origin information. The *dbarrival* object is extracted from the *dbarrivals* container for this station as shown in the inner-most loop in Figure 26 on page 204. The results of processing the arrival are written to the database tables, which can include: **arrival**, **apma**, **amplitude**, and **amp3c**. The exact tables depend on the station and phase types.



TABLE 105: FUNCTIONAL INTERFACE OF PROCESS-RECALL-ARRIVAL

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>dbarrival</i>	input	GObj	<i>dbarrival</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

For more information about this function, see “Detailed Description of Perform Recall Signal Processing [5.4].”

### Detailed Description of Perform Recall Signal Processing [5.4]

Recall processing is implemented by the function `process-recall-arrival`. The architecture of the process is shown in Figure 27. Within the process, six subprocesses are executed sequentially: waveforms are acquired (5.4.1), a *detection* object is created (5.4.2), the highest snr beam is found and stored in the *detection* object (5.4.3), the detection attributes are created or revised (5.4.4), primary detection measurements are screened (5.4.5), and finally, secondary features are measured, and all of the detection attributes are written to the database (5.4.6).

The following subprocesses shown in Figure 27 are described in this section:

- *Acquire Waveforms [5.4.1]*
- *Create Detection [5.4.2]*
- *Identify Best Beam [5.4.3]*
- *Revise Detection Attributes [5.4.4]*
- *Screen Detections [5.4.5]*
- *Estimate Arrival Features and Write to Database [5.4.6]*

## ▼ Seismic Processing

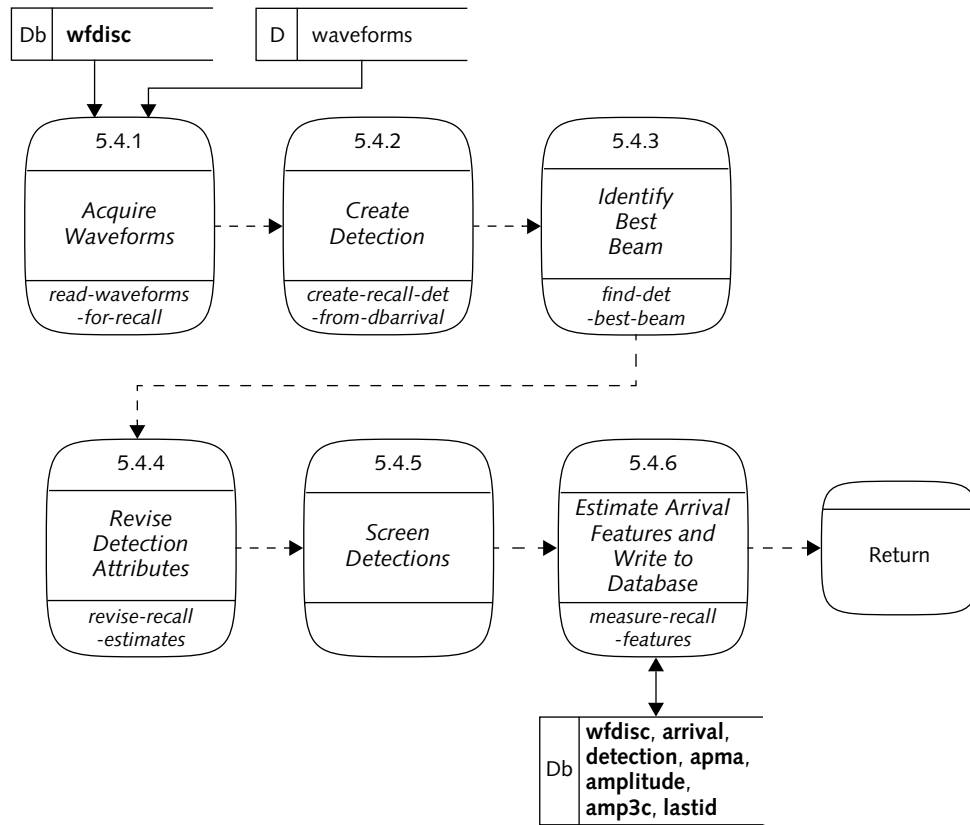


FIGURE 27. ARCHITECTURE OF PERFORM RECALL SIGNAL PROCESSING

**Acquire Waveforms [5.4.1]**

This subprocess acquires waveforms that are used to make feature measurements. It is implemented by the local Scheme function `read-waveforms-for-recall`. Acquiring waveforms is a common procedure in many *DFX* Scheme applications. The purpose is to generate *wfdata* and *wfmem* containers that store the waveforms for the current arrival at the current station. For general information on acquiring waveforms see “Reading Waveforms” on page 58.

The waveforms container *wfdata-con* is filled by the Scheme code:

```
(set! wfdata-con (read-waveforms-for-recall initsite
      dbarrival))
```

The function requires two arguments, which are described in Table 106. The data are returned in a *wfdata* container that covers a specific arrival at a single seismic station (but may contain multiple channels). The time interval is determined from the detection recipe and the arrival time of the current detection.

**TABLE 106: FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-RECALL**

Argument Name	Usage	Type	Description
<i>initsite</i>	input/output	GObj	<i>initsite</i> object
<i>dbarrival</i>	input	GObj	<i>dbarrival</i> object
n/a	returned value	container	<i>wfdata</i> objects

#### Create Detection [5.4.2]

This subprocess creates a *detection* object, which is used throughout the computations to hold various signal attributes. It is created using the information from the *dbarrival* object by the global Scheme function `create-recall-det-from-dbarrival`. The function copies the matching attributes from the *dbarrival* object into a *detection* object allowing for any necessary unit conversions. This function is called by the Scheme language code:

```
(set! det (create-recall-det-from-dbarrival dbarrival))
```

The function requires one input argument, *dbarrival*, which is the current *dbarrival* object. The returned value is a *detection* object.

## ▼ Seismic Processing

**Identify Best Beam [5.4.3]**

This subprocess finds the highest-snr beam out of all the beams in the site-specific beam recipe file. It implemented by the local Scheme function `find-det-best-beam`. The function requires the same structure as the *Identify Best Beam* subprocess in *Interactive Seismic Recall*. For more information see “Identify Best Beam [4.4.3]” on page 196.

**Revise Detection Attributes [5.4.4]**

This subprocess performs the bulk of the feature measurements via the local Scheme function `revise-recall-estimates`. These features include: (i) uncertainty in arrival time, (ii) amplitude for the “best” beam and the f-k beam, (iii) azimuth and its uncertainty, and (iv) slowness and its uncertainty. The function is called by the Scheme language code:

```
(revise-recall-estimates wfdata-con initsite det)
```

The function has the same structure as the *Revise Detection Attributes* subprocess in *Interactive Seismic Recall*. For more information see “Revise Detection Attributes [4.4.4]” on page 198.

**Screen Detections [5.4.5]**

This subprocess screens estimates of primary detection attributes. It is implemented by the local Scheme function `screen-recall-detection`, which is a wrapper function to the global Scheme function `screen-detections`. The function `screen-detections` compares the estimated values of primary features to screening limits. Because only one detection is passed into the function in this application, only feature values subject to screening are checked. No new feature measurements are recorded for screened detections. The function is called by the Scheme language code:

```
(screen-recall-detection initsite det)
```

The function requires two arguments, which are described in Table 107.

TABLE 107: FUNCTIONAL INTERFACE OF SCREEN-RECALL-DETECTIONS

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>det</i>	input and output	GObj	<i>detection</i> object
n/a	returned value	GObj	<i>detection</i> object (null value indicates error)

#### Estimate Arrival Features and Write to Database [5.4.6]

This subprocess creates and fills database objects that are then written to the appropriate database tables. It is implemented by the local Scheme function `measure-recall-features`. The function follows the same basic algorithm as the *Interactive Seismic Recall* application with the addition of updating the slowness and azimuth residuals if not previously set. For more information see “Estimate Arrival Features and Write to Database [4.4.5]” on page 200.

## AUTOMATIC DEPTH-PHASE SNR [6]

*Automatic Depth-phase SNR (DPSNR)* computes a specialized snr value for seismic depth phases. The depth-phase snr measure is a ratio of the amplitude in a short interval including and following the arrival to the amplitude in a short interval preceding the arrival. *DPSNR* writes this snr value to the **amplitude** table. The depth-phase snr should not be confused with the standard *snr* value in the **amplitude** table, which is the ratio of a detection's maximum short-term average to the long-term average. The application source code resides in the file `DFX-depth-phase-snr.scm`. *DPSNR* is normally run as a post-analysis process.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Estimate Depth-phase SNR* [6.3]

### General Description

*DPSNR* was designed to measure the snr of depth phases. The snr computation attempts to quantify how distinguishable the depth phase is from the surrounding coda. *Event Screening* uses this value as part of its criteria in deciding the confidence in the depth-phase identification and the computed event depth.

## Architecture

Figure 28 shows the architecture of *DPSNR* with its subordinate processes. The first process retrieves the appropriate depth phases from the database (6.1). The principal computational process is located within a pair of nested loops: the outer loop over stations, and the inner loop over arrivals. The outer (station) loop first extracts station-specific parameters and database information (6.2) and then invokes the inner (arrival) loop to perform the measurements. The inner process (6.3) reads the waveforms, computes the snr value, and writes the amplitude records.

## Input/Output

Input processing parameters are obtained from the CVAR datastore. The input parameters used by this process are described in Table 108. The database tables read by this application are indicated in Table 229 on page 443.

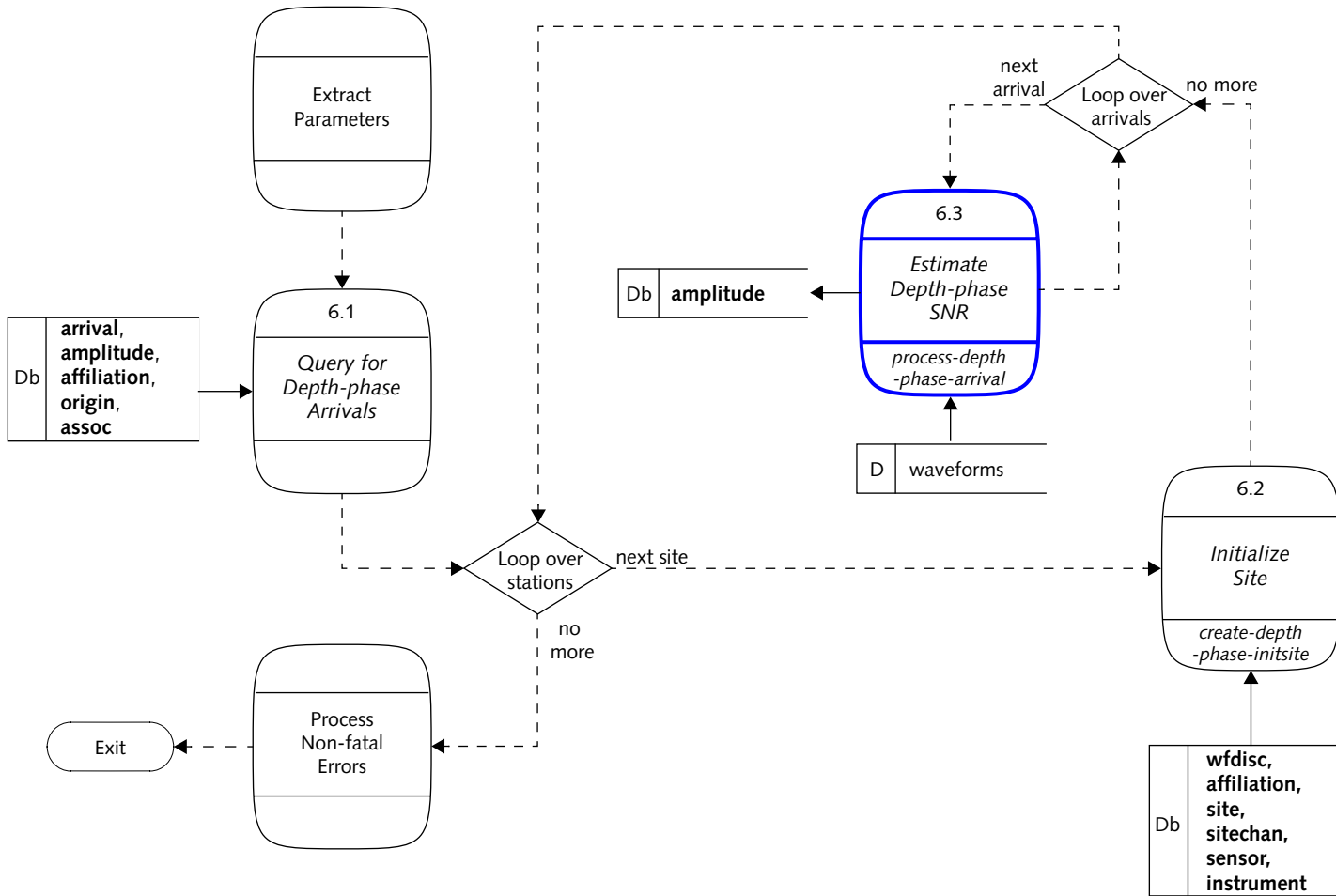


FIGURE 28. ARCHITECTURE OF DPSNR



**TABLE 108: INPUT PARAMETERS FOR DEPTH-PHASE SNR PROCESSING**

Par Name	Typical Usage	Default Value	Description
<i>net</i>	command line	<none>	network of stations to process
<i>start-time</i>	command line	<none>	epoch time to use for the start-time of the interval containing origins to process
<i>end-time</i>	command line	<none>	end-time of the processing interval
<i>out-amplitude-table</i>	par file	<b>amplitude</b>	output table containing amplitude information generated by this application
<i>affiliation-table</i>	par file	<b>affiliation</b>	table containing station and network affiliations
<i>arrival-table</i>	par file	<b>arrival</b>	input <b>arrival</b> table
<i>assoc-table</i>	par file	<b>assoc</b>	input <b>assoc</b> table
<i>instrument-table</i>	par file	<b>instrument</b>	table containing generic (default) calibration information about a station
<i>origin-table</i>	par file	<b>origin</b>	input <b>origin</b> table
<i>sensor-table</i>	par file	<b>sensor</b>	table containing specific calibration information for physical channels
<i>site-table</i>	par file	<b>site</b>	table containing site-location information
<i>sitechan-table</i>	par file	<b>sitechan</b>	table containing station-channel information
<i>in-wfdisc-table</i>	par file	<b>wfdisc</b>	input table containing waveform file header and descriptive information

## ▼ Seismic Processing

**TABLE 108: INPUT PARAMETERS FOR DEPTH-PHASE SNR PROCESSING (CONTINUED)**

Par Name	Typical Usage	Default Value	Description
<i>beam-filter-onset-len</i>	par file	100.0	time interval in seconds preceding and following analysis interval provided to allow the filter to stabilize
<i>data-offset-len</i>	par file	50.0	parameter (units = seconds) used to encompass any expected edge effects from subsequent processing
<i>depth-phase-site-recipe-file</i>	par file	<none>	parameter file that loads site-specific recipes
<i>depth-phase-snr-amprec-list</i>	par file	<none>	list of amplitude recipes to use, in the form "amprec1,amprec2,amprec3..."
<i>depth-phase-snr-amptype</i>	par file	dp-snr	string to use as the <i>amp-type</i> attribute in the output amplitude record
<i>recall-depth-phases</i>	par file	pP, sP, pPKP, pPKPbc, pPKiKP	list of phases to recognize as depth phases
<i>wfdisc-extension-len</i>	par file	86400.0	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem

Table 109 lists the data produced by this application. The *snr* values are written to the **amplitude** table. Table 235 on page 447 lists the specific attributes written by this application.

TABLE 109: DATA PRODUCED BY DEPTH-PHASE SNR PROCESSING

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>amplitude</b>	database table	amplitude values for depth phases

## Processes

The following processes shown in Figure 28 on page 220 are described in this section:

- *Query for Depth-phase Arrivals [6.1]*
- *Initialize Site [6.2]*
- *Estimate Depth-phase SNR [6.3]*

### Query for Depth-phase Arrivals [6.1]

This process retrieves the depth-phase arrivals for which snr values will be estimated. It is implemented by the local Scheme function `query-for-dbarrival-depth-phases`. The function obtains the arrival set that satisfies the following constraints: (i) the arrivals are associated with origins in the specified input time interval, (ii) the arrival's phase identifier is one of the specified depth-phases, and (iii) the arrivals do not already have a depth-phase snr measurement. This function is called by the Scheme language code:

```
(set! dbarrival-con (query-for-dbarrival-depth-phases))
```

This function requires no arguments. It uses the internal parameters described in Table 110 obtained from the CVAR datastore. The function returns a *dbarrival* container satisfying the constraints described in the previous paragraph. For information on database queries see "Database Operations" on page 40.

## ▼ Seismic Processing

**TABLE 110: INTERNAL SYMBOLS FOR QUERY-FOR-DBARRIVAL-DEPTH-PHASES**

Symbol	Type	Description
<i>net</i>	character-string	network name
<i>start-time</i>	real	epoch start-time of processing interval
<i>end-time</i>	real	end-time of the processing interval
<i>affiliation-table</i>	character-string	table containing station and network affiliations
<i>out-amplitude-table</i>	character-string	output table containing amplitude information generated by this application
<i>arrival-table</i>	character-string	<b>arrival</b> table name
<i>assoc-table</i>	character-string	<b>assoc</b> table name
<i>origin-table</i>	character-string	<b>origin</b> table name
<i>depth-phase-snr-amptype</i>	character-string	name used to identify a depth-phase snr measure
<i>recall-depth-phases</i>	character-string	list of recognized depth phases

**Initialize Site [6.2]**

This process obtains station-specific processing parameters from the configuration recipe files and site-related database tables. After the information has been collected it is returned in an *initsite* object. Table 8 on page 50 lists the entities contained in the *initsite* object. The *initsite* object is created via the station-driven method described in “Initializing Sites” on page 46. The process is implemented by the local Scheme function `create-depth-phase-initsite`. This function is called by the Scheme language code:

```
(set! initsite (create-depth-phase-initsite sta
dbarrival-con))
```

The function requires two arguments, which are described in Table 111.

**TABLE 111: FUNCTIONAL INTERFACE OF CREATE-DEPTH-PHASE-INITSITE**

Argument Name	Usage	Type	Description
<i>sta</i>	input	character-string	station code
<i>dbarrival-con</i>	input	container	<i>dbarrival</i> objects
n/a	returned value	GObj	<i>initsite</i> object

### Estimate Depth-phase SNR [6.3]

This process performs the main computation tasks for estimating the snr and writes the computed values to the database. It performs the following steps: acquire waveforms, measure amplitude and snr values, find the best quality measurement, and finally, write the snr value to the **amplitude** table.

The processing occurs within the inner loop over arrivals (see Figure 28 on page 220). The function measures the depth-phase snr on a single arrival via the local Scheme function `process-depth-phase-arrival`. It is called by the Scheme language code:

```
(set! dbarrival-con (process-depth-phase-arrival initsite
  dbarrival))
```

This function requires two arguments, which are described in Table 112. The next section provides more information about this function.

**TABLE 112: FUNCTIONAL INTERFACE OF PROCESS-DEPTH-PHASE-ARRIVAL**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>dbarrival</i>	input	GObj	<i>dbarrival</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

### Detailed Description of Estimate Depth-phase SNR [6.3]

Figure 29 shows the architecture of this process, which consists of five subprocesses. The first subprocess acquires the waveforms for the input arrival (6.3.1). Next, a set of amplitude recipes are retrieved from the *initsite* object. The amplitude recipes cover several frequency bands, and each frequency band has two recipes; one defines the signal amplitude measure, and the other defines the pre-signal amplitude measure. The process enters a loop over all of the amplitude recipes. Each recipe is fed into the *DFX/libamp* function *measure-detection-amplitude* (6.3.2). The amplitude measures are then saved in a *dbamplitude* object. The next subprocess pairs the amplitude measures (pre-signal and signal) for each frequency band (6.3.3). Next, the amplitudes in a frequency band that produce the greatest snr value are chosen as the best pair (6.3.4), and that pair is used to form the final *dbamplitude* object. The object is screened to ensure that it does not already exist in the database, and finally it is written to the **amplitude** table (6.3.5).

The following subprocesses shown in Figure 29 are described in this section:

- *Acquire Waveforms [6.3.1]*
- *Measure Amplitudes [6.3.2]*
- *Make Amplitude Pairs [6.3.3]*
- *Find Best Pair [6.3.4]*
- *Make Amplitude Record [6.3.5]*
- *Write to Database [6.3.6]*

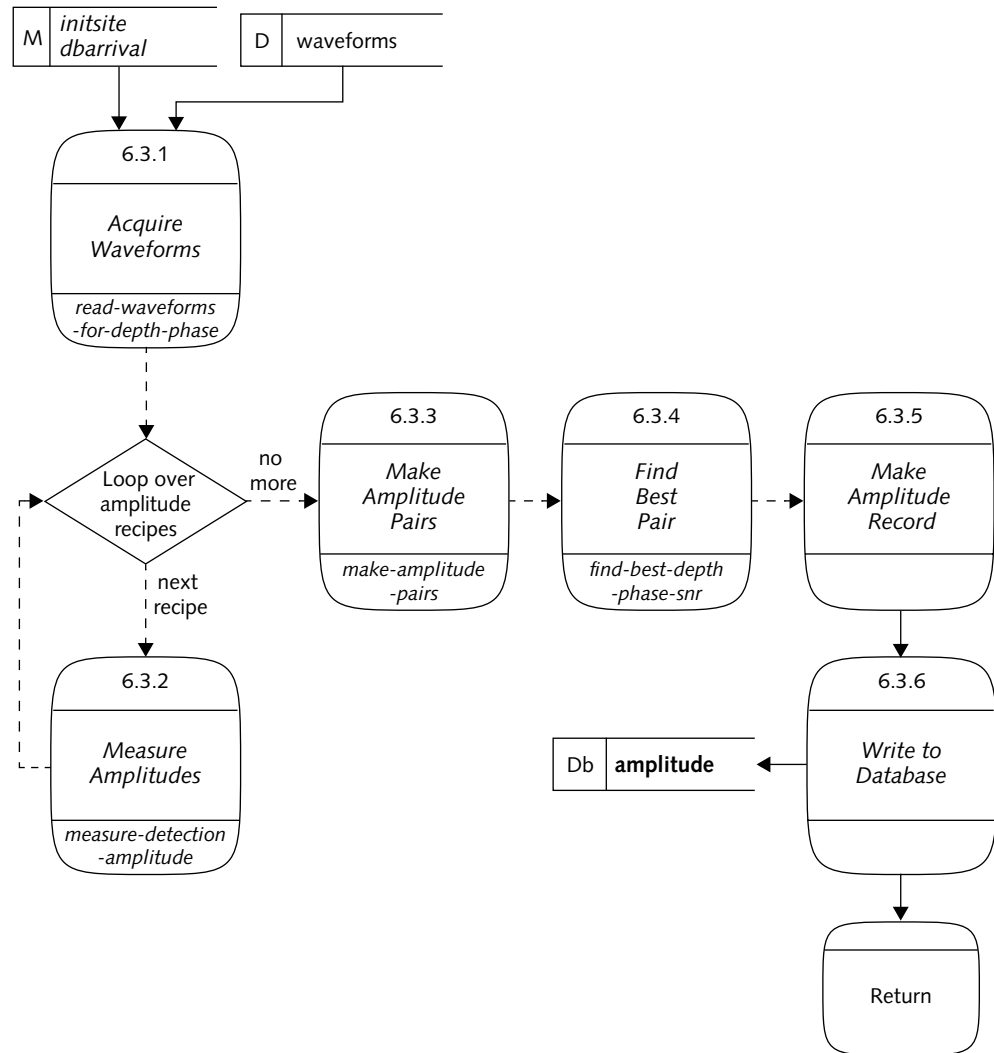


FIGURE 29. ARCHITECTURE OF PERFORM ESTIMATE DEPTH-PHASE SNR

**Acquire Waveforms [6.3.1]**

This subprocess obtains waveforms surrounding the time of the input arrival. The waveforms are used to make the amplitude measurements. They are acquired by a local Scheme function, which is called by the code:

## ▼ Seismic Processing

```
(set! wfddata-con (read-waveforms-for-depth-phase initsite
dbarrival))
```

This function requires two arguments, which are described in Table 113.

**TABLE 113: FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-DEPTH-PHASE**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>dbarrival</i>	input	GObj	<i>dbarrival</i> object
n/a	returned value	container	<i>waveform</i> objects

Within the function the waveforms are read and set into the *wfmem* container through a call to *DFX/libdata* function *read-waveforms*, and their quality is checked by the *DFX/libqc* function *qc-waveforms*. The populated *wfddata* container is returned to the calling function *process-depth-phase-arrival*. For more information see “Reading Waveforms” on page 58.

### Measure Amplitudes [6.3.2]

This process computes a set of signal and noise amplitudes over a variety of frequency bands as specified by entries in the amplitude recipe. The *DFX/libamp* function *measure-detection-amplitude* is located within a loop over amplitude recipes. The amplitude recipes are obtained from the *initsite* object. The amplitude measure function is called by the Scheme code:

```
(set! amp (measure-detection-amplitude wfddata-con amprec
det))
```

The function requires three arguments, which are described in Table 114.



TABLE 114: FUNCTIONAL INTERFACE OF MEASURE-DETECTION-AMPLITUDE

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	waveforms
<i>amprec</i>	input	GObj	amplitude recipe parameters
<i>det</i>	input	GObj	<i>detection</i> object
n/a	returned value	GObj	<i>amplitude</i> object

The function returns an amplitude object. A *dbamplitude* object is created from the returned *amplitude* object and the input *detection* object by the global Scheme function `create-detection-dbamplitude-from-amp`. The *dbamplitude* objects are collected in a container for use in subsequent processing.

### Make Amplitude Pairs [6.3.3]

This process is implemented by the local Scheme function `make-amplitude-pairs`. This function creates pairs of signal and noise measurements for the specified frequency bands so an snr measure can be computed. The signal and noise amplitudes are matched by the *chan* attribute value, which contains the beam recipe name used in the amplitude measurement. Within a loop over each beam recipe, the *dbamplitude* objects with *chan* attributes matching the beam recipe name are extracted from the *dbamplitude* container. There are two measurements, one for the signal and one for the noise, for each beam. An error is printed if this not be the case.

The signal and noise measurements are identified by their measurement times; the noise measurements occur before the signal measurements. The pair is appended to a list of pairs, which is returned to the calling function `find-best-depth-phase-snr` (a local Scheme function).

## ▼ Seismic Processing

**Find Best Pair [6.3.4]**

This process is implemented by the local Scheme function `find-best-snr-pair`. The function identifies the signal and noise pair with the best ratio of signal amplitude to noise amplitude. First, measurement pairs without a valid noise measurement are removed from the list. Next, a loop over the surviving pairs finds the highest snr pair, and that list of paired *amplitude* objects is returned to the calling function `find-best-depth-phase-snr` (a local Scheme function).

**Make Amplitude Record [6.3.5]**

This process occurs within the body of the local Scheme function `find-best-depth-phase-snr`. The function uses the data from the best snr amplitude pair to create a single *dbamplitude* object. The *dbamplitude* object is returned by the local Scheme function `find-best-snr-pair`. The final amplitude record attributes are initially set to the signal amplitude attributes of the best pair. The final amplitude *snr* attribute is set to the ratio of the signal and noise *amp* attributes. The *amptype* attribute is set to the value of the *depth-phase-snr-amptype* parameter, defaulting to the value `dp-snr`. The *per* attribute is set to the inverse of the amplitude-beam filter-band center frequency, and the *bandw* attribute is set to the width of the beam filter band. The *dbamplitude* object is returned by `find-best-snr-pair` to the calling function `make-depth-phase-amplitude` (a local Scheme function), which sets it into a suitable container. This latter function then returns the container with the best *dbamplitude* object to the calling function `measure-depth-phase-snr` (a local Scheme function). This latter function invokes screening and submission to the database (the activities of the database are described in the next section).

**Write to Database [6.3.6]**

This process occurs within the local Scheme function `measure-depth-phase-snr`. The container with the depth-phase amplitude measurement, a *dbamplitude* object, is sent to the function `screen-arrival-dbamplitude-by-existing` to preclude duplicate **amplitude** table entries (see “Screening Database Containers” on page 43). If the *dbamplitude* object has not been removed by screening, it is

written to the database using the global Scheme function `submit-container-db`. If a submission is successful, the insert is committed by the global Scheme function `commit-db`; otherwise, an error message is printed. For detailed information about `submit-container-db` see “Writing to the Database” on page 44.

## AUTOMATIC NOISE AMPLITUDE ESTIMATION [7]

*Automatic Noise Amplitude Estimation (NoiseAmp)* is a DFX Scheme application that produces amplitude estimates for theoretical arrivals for a given time interval and network. The application source code resides in the file `DFX-noiseamp.scm`. For each origin in the interval, *NoiseAmp* produces noise amplitude estimates for stations without associated arrivals. Measurements are made at the predicted arrival times, azimuths, and slownesses for the given origin. The results of *NoiseAmp* are written to the **amplitude** and **parrival** tables.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Estimate Noise Amplitude* [7.5]

### General Description

*NoiseAmp* determines the maximum possible signal level at stations that do not have an observed arrival. These measures can be used in Threshold Monitoring and in generating maximum-likelihood-estimation (MLE) magnitude estimates. *NoiseAmp* was designed to operate on a network of stations within a time interval defined by a set of origins. This suggests the principal architectural features of the application: a pair of nested loops consisting of an outer loop over origins and an inner loop over stations. Interior to these loops lies the computational core of the application, the process *Estimate Noise Amplitude*. Before this process can be run a variety of initialization activities must occur.

## Architecture

This application is implemented by a pair of nested loops: the outer loop over origins and the inner loop over sites, as show in Figure 30. The first initialization process retrieves three key parameters from the CVAR datastore: *start-time*, *end-time*, and *origin-table*. These define the processing interval to be searched and the name of the database table in which to search for origins. The next two initialization processes identify the relevant origins and develop the site-specific processing information for these origins. After the three initialization processes are complete the application proceeds to the pair of nested loops. The inner loop contains two processes, which extract parameters and estimate noise amplitude. Prior to invoking the nested loops in which the principal *NoiseAmp* processing is called, the following processes are implemented: (i) initialize parameters, which are the key parameters from the datastore, (ii) query the database, which is accomplished by the local Scheme function `query-for-dborigins-by-time`, and returns a *dborigins* container, and (iii) initiate sites, which is accomplished by the local Scheme function `initialize-net-sites`, and returns an *initsite* container. The *dborigin* objects identify the origins for which *NoiseAmp* processing is to be performed. The *initsite* objects contain the site-specific information for the stations that are affiliated with the network on which the origins were developed.

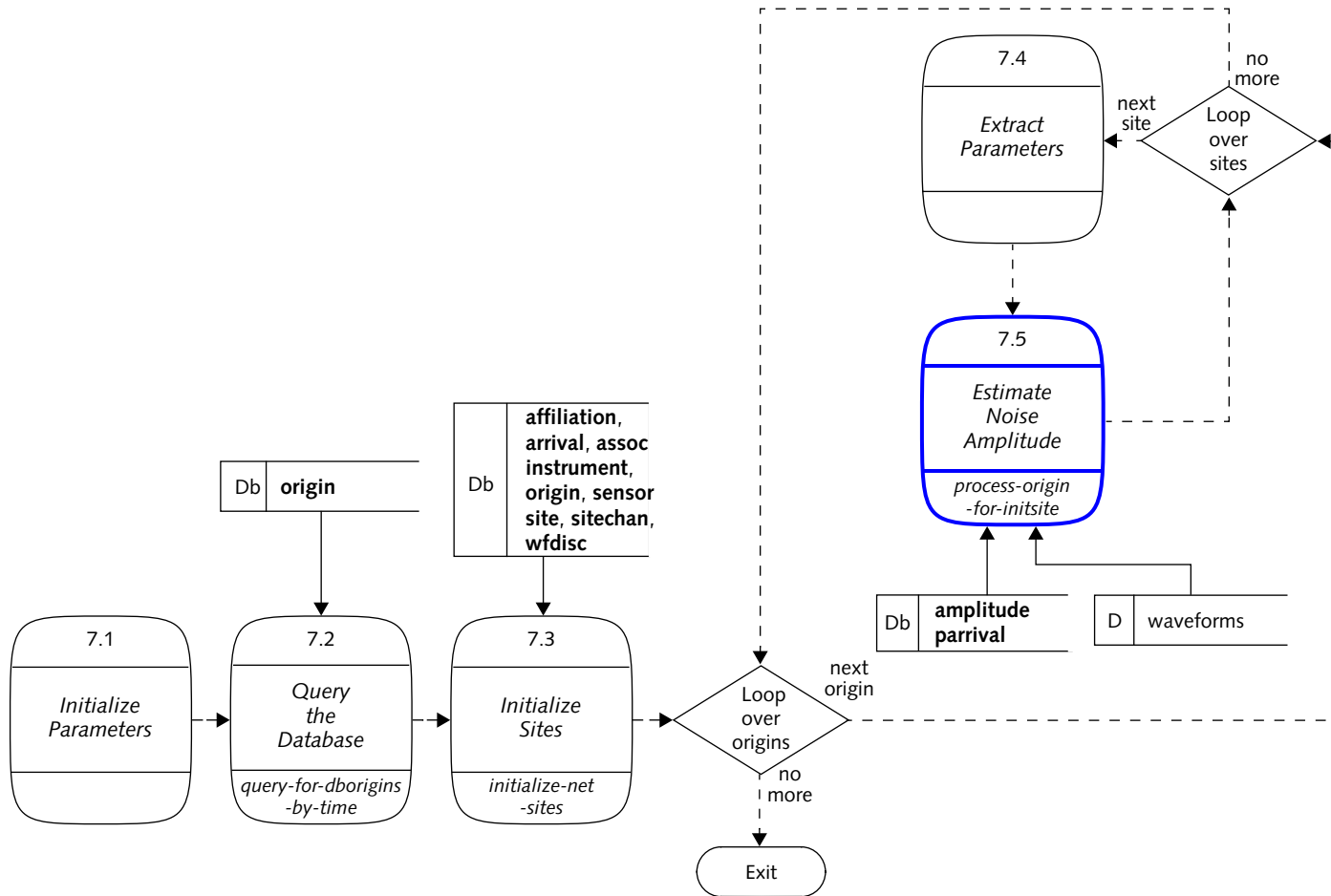


FIGURE 30. ARCHITECTURE OF NOISEAMP

## Input/Output

*NoiseAmp* is called for a given network and time interval. The network code, start-time, and end-time parameters are generally specified on the command line. The rest of the parameters are set or referenced in the `DFX-noiseamp.par` file. The input CVAR datastore parameters used in the Scheme code are described in Table 115.

**TABLE 115: INPUT PARAMETERS FOR AUTOMATIC NOISE AMPLITUDE ESTIMATION**

Par Name	Typical Usage	Default Value	Description
<i>net</i>	command line	<none>	network of stations to process
<i>start-time</i>	command line	<none>	start-time of the processing interval
<i>end-time</i>	command line	<none>	end-time of the processing interval
<i>noiseamp-max-delta</i>	par file	360.0	maximum distance to stations (degrees)
<i>noiseamp-min-delta</i>	par file	0.0	minimum distance to stations (degrees)
<i>noiseamp-data-off-set-len</i>	par file	<none>	extra data required to ensure that <i>NoiseAmp</i> processing will be free of edge effects
<i>noiseamp-site-recipe-file</i>	par file	<none>	file containing site-specific processing recipes
<i>noiseamp-data-tirec-list</i>	par file	<none>	list of time-interval recipes
<i>noiseamp-theoretical-amprec-list</i>	par file	<none>	list of amplitude estimation processing recipes for the theoretical arrivals

## ▼ Seismic Processing

**TABLE 115: INPUT PARAMETERS FOR AUTOMATIC NOISE AMPLITUDE ESTIMATION (CONTINUED)**

Par Name	Typical Usage	Default Value	Description
<i>wfdisc-extension-len</i>	par file	86400.0	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>affiliation-table</i>	par file	affiliation	table containing station and network affiliations
<i>instrument-table</i>	par file	instrument	table containing generic (default) calibration information about a station
<i>origin-table</i>	par file	origin	table containing origin information
<i>sensor-table</i>	par file	sensor	table containing specific calibration information for physical channels
<i>site-table</i>	par file	site	table containing site-location information
<i>sitechann-table</i>	par file	sitechann	table containing station-channel information
<i>in-arrival-table</i>	par file	arrival	input table containing arrival information
<i>in-assoc-table</i>	par file	association	input table containing association information
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information
<i>out-amplitude-table</i>	par file	amplitude	output table containing amplitude information generated by this application
<i>out-parrival-table</i>	par file	parrival	output table containing predicted arrival information generated by this application



All results from this application are written to the database tables described in Table 116.

**TABLE 116: DATA PRODUCED BY AUTOMATIC NOISE AMPLITUDE ESTIMATION**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>amplitude</b>	database table	predicted arrival amplitudes
<b>parrival</b>	database table	predicted arrivals

Table 236 on page 448 lists the database attributes written by *NoiseAmp*.

## Processes

The following processes shown in Figure 30 on page 234 are described in this section:

- *Initialize Parameters [7.1]*
- *Query the Database [7.2]*
- *Initialize Sites [7.3]*
- *Extract Parameters [7.4]*
- *Estimate Noise Amplitude [7.5]*

### Initialize Parameters [7.1]

To define the processing interval and source of origin information, this process acquires three parameters *t1*, *t2*, and *origin-table* via access to the CVAR datastore using the global Scheme functions `mstpar` and `mstspar`. These parameters are the start-time, end-time, and the database's origin table name, respectively.

## ▼ Seismic Processing

**Query the Database [7.2]**

To acquire the required origins, this process queries the database for origins in the given time interval via the local Scheme function `query-for-dborigins-by-time`. For general information on how database queries are implemented see “Querying the Database” on page 41. The function `query-for-dborigins-by-time` requires three arguments: the foregoing symbols *t1*, *t2*, and *origin-table*. This function returns a *dborigin* container with objects that have the same structure as the **origin** database table [IDC5.1.1Rev2].

**Initialize Sites [7.3]**

A variety of site-specific information is required to support *NoiseAmp*. This process extracts and generates the appropriate collection of information and returns it in an *initsite* container. This is a special case of the Initialize Site function (see “Initializing Sites” on page 46). This process is implemented by the local Scheme function `initialize-net-sites`. The function `initialize-net-sites` is called by the Scheme code:

```
(set! initsite-con (initialize-net-sites dborigin-con))
```

This function requires one argument, which is described in Table 68 on page 155.

This function returns an *initsite* container for all origins and the appropriate stations given the processing time interval, network, and the set of origins to process. The seventh column of Table 8 on page 50 lists those entities contained in the *initsite* objects created by this function for *NoiseAmp*.

The function `initialize-net-sites` particular to *NoiseAmp* rejects stations having arrivals associated with the origin being processed. This rejection step in the *initsite* creation process ensures that the downstream processing is concerned only with those stations that did not detect an arrival from the origin.

**Extract Parameters [7.4]**

This process acquires both the site-dependent and the origin-dependent parameters described in Table 117. This process is implemented by the inline common *libgobj* Scheme function `extract-gobj-attr`.

**TABLE 117: INTERNAL SYMBOLS FOR PROCESS-ORIGINS**

Symbol	Type	Description
<i>sta</i>	character-string	station code of the station for which processing is being performed
<i>orid</i>	integer	unique identification number of the origin for which processing is being performed

After these parameters are acquired, the principal computational process, *Estimate Noise Amplitude*, is called.

**Estimate Noise Amplitude [7.5]**

The process *Estimate Noise Amplitude* performs the bulk of the *NoiseAmp* computational processing. It is implemented by the local Scheme function `process-origin-for-init-site`. The function `process-origin-for-init-site` is called by the Scheme code:

```
(process-origin-for-init-site init-site origin)
```

The function requires two arguments, which are described in Table 118.

**TABLE 118: FUNCTIONAL INTERFACE OF PROCESS-ORIGIN-FOR-INITSITE**

Argument Name	Usage	Type	Description
<i>init-site</i>	input	GObj	<i>init-site</i> object
<i>origin</i>	input	GObj	<i>dborigin</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

## ▼ Seismic Processing

The function `process-origin-for-init-site` performs the following steps:

1. Determine whether or not the theoretical arrival amplitudes need to be estimated for a given origin and station, and if so, acquire the waveforms.
2. Estimate the amplitude.
3. Write the results to the database.

For more information about this function, see “Detailed Description of Estimate Noise Amplitude [7.5].”

### Detailed Description of Estimate Noise Amplitude [7.5]

This process contains the key elements of the *NoiseAmp* application. For each origin and station, it ensures that the event-to-station distance is acceptable. If so, it gets the waveforms, estimates the amplitude, and saves the results in the database.

This process is implemented by the local Scheme function `process-origin-for-init-site`. Figure 31 shows the architecture of the process, which consists of a single pass through six subprocesses. The first subprocess extracts parameters for the given station and origin. The second and third subprocesses check the event-to-station distance and extract the associated arrivals for this origin if any exist. If no associated arrivals exist, the fourth subprocess acquires waveforms. The last two subprocesses estimate origin amplitudes for the predicted arrivals (that is, theoretical arrivals) and write the estimates for the predicted arrivals to the database. The process *Estimate Noise Amplitude* also uses a local Scheme function, `submit-aux`, to write the results to the database and roll back the database if the submission is unsuccessful.

If the event-to-station distance fails to satisfy the distance tests, control is returned to the calling program without further processing. If the event-to-station distance is inside the acceptable distance interval, this process produces the estimate of noise amplitude.

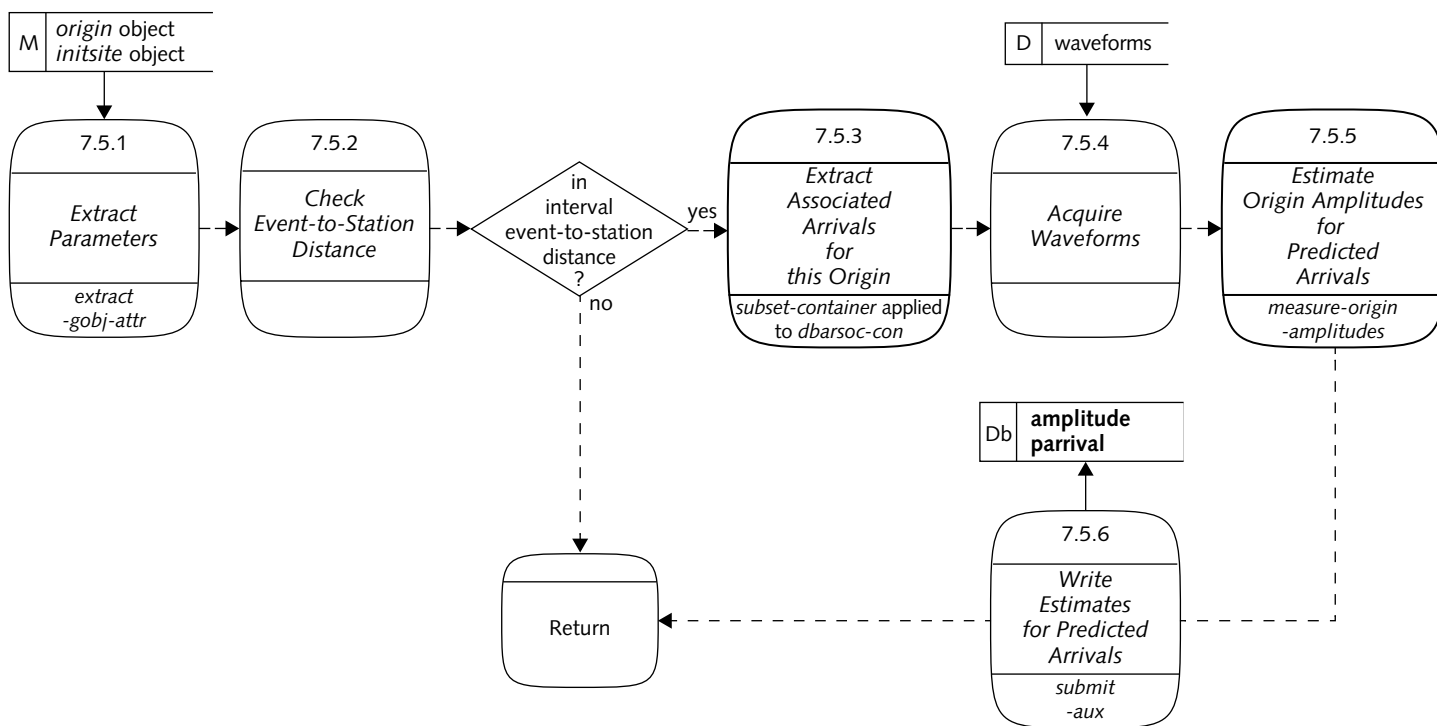


FIGURE 31. ARCHITECTURE OF ESTIMATE NOISE AMPLITUDE

## ▼ Seismic Processing

The following subprocesses shown in Figure 31 are described in this section:

- *Extract Parameters [7.5.1]*
- *Check Event-to-Station Distance [7.5.2]*
- *Extract Associated Arrivals for this Origin [7.5.3]*
- *Acquire Waveforms [7.5.4]*
- *Estimate Origin Amplitudes for Predicted Arrivals [7.5.5]*
- *Write Estimates for Predicted Arrivals [7.5.6]*

### Extract Parameters [7.5.1]

This subprocess acquires and sets values for the local symbols used to perform the process *Estimate Noise Amplitude*. Inline Scheme code, via the intrinsic functions `let*` and `car`, the global Scheme function `getpar`, and the common *libgobj* Scheme function `extract-gobj-attr`, is used to acquire values, symbols, objects, and containers of objects for the station and origin to be processed for this time interval. In addition, the distance (symbol *delta*) from the event to the station is extracted by `car` from the first element of the list returned by the *DFX/libutil* function `compute-distance-azimuth`, which calculates the distance and azimuth. The specific parameters that are extracted are described in Table 119.

**TABLE 119: INTERNAL SYMBOLS FOR EXTRACT PARAMETERS**

Symbol	Type	Description
<i>amplitude-table</i>	character-string	name of the <b>amplitude</b> database table
<i>parrival-table</i>	character-string	name of the <b>parrival</b> database table
<i>orid</i>	integer	unique identification number for the origin being processed
<i>sta</i>	character-string	station code of the station for which processing is being performed
<i>site</i>	GObjs of <i>dbsite</i> class	object with site-specific information for the site being processed
<i>tirec-con</i>	<i>time-interval recipe</i> container	time-interval recipes used to estimate the noise amplitudes

TABLE 119: INTERNAL SYMBOLS FOR EXTRACT PARAMETERS (CONTINUED)

Symbol	Type	Description
<i>beamrec-con</i>	<i>beam recipe</i> container	recipes to govern the nature of the beams to be formed (type, direction, threshold, spectral filtering, and so on)
<i>amprec-con</i>	<i>amplitude estimation processing recipe</i> container	amplitude estimation processing recipes
<i>arsoc-con</i>	<i>arsoc</i> container	associated-arrivals
<i>wfdata-con</i>	<i>wfdata</i> container	initially empty container of information about waveforms
<i>dbarsoc-con</i>	<i>dbarsoc</i> container	initially empty container of information about associated-arrivals
<i>delta</i>	real	event-to-station distance (degrees)
<i>min-delta</i>	real	minimum event-to-station distance for <i>NoiseAmp</i> processing
<i>max-delta</i>	real	maximum event-to-station distance for <i>NoiseAmp</i> processing

**Check Event-to-Station Distance [7.5.2]**

Noise amplitude estimates are generally only useful where the event-to-station distance lies in a suitable distance interval. This subprocess determines whether this is the situation. Inline Scheme code, via the intrinsic functions `if` and `or`, is used to determine whether the site-to-event distance (symbol *delta*) is inside the bounds set by the local symbols *min-delta* and *max-delta*, which were extracted from the CVAR datastore (see previous paragraph). If *delta* is in bounds, processing continues to the next subprocess, *Extract Associated Arrivals for this Origin*. If *delta* is out of bounds a warning message is printed and control is returned to the calling program.

## ▼ Seismic Processing

**Extract Associated Arrivals for  
this Origin [7.5.3]**

This subprocess extracts the existing arrivals associated with the given origin. If associated arrivals exist at this station then no further processing is required. Inline Scheme code, via the intrinsic functions `if` and `not` as well as global Scheme functions `subset-container` and `container-empty?`, is used to extract existing associated arrivals from the associated arrivals container referenced by the *initsite* object. Where the associated arrivals match the current *orid* value and station code (symbols *orid* and *sta*), respectively, a warning message is returned, no further processing is performed in this subprocess, and control is returned to the calling program. Otherwise, the process *Estimate Noise Amplitude* transfers control to the next subprocess, *Acquire Waveforms*.

**Acquire Waveforms [7.5.4]**

This subprocess reads the waveforms for the requisite time interval and stations. It also invokes quality-control processing to evaluate and perform minor repairs on the waveforms. Inline Scheme code, via the intrinsic functions `set!` and `if`, the global Scheme function `container-empty?`, and the local Scheme function `read-waveforms-for-initsite`, is used to acquire the waveforms for this origin and *initsite*. The function `read-waveforms-for-initsite` is called by the Scheme code:

```
(set! wfdata-con (read-waveforms-for-initsite initsite
origin))
```

The function requires two arguments, which are described in Table 120.

**TABLE 120: FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-INITSITE**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>origin</i>	input	GObj	<i>origin</i> object
n/a	returned value	container	<i>wfdata</i> objects



The function `read-waveforms-for-initsite` performs the following steps:

1. Initialize some local symbols.
2. Check that neither of its input arguments is null.
3. Make a local copy of the container of waveforms referenced by the *initsite* object.
4. Extract the appropriate subset of time interval recipes for the waveforms to be acquired.
5. Determine the time interval for processing based on the station-origin distance including extra waveforms for edge effects introduced by beamforming.
6. Read the actual waveforms required from the *wfmem* container referenced by the *initsite* object via the *DFX/libdata* function `read-waveforms`.
7. Evaluate and perform minor repairs on the waveforms via the *DFX/libqc* function `qc-waveforms`.

The function `read-waveforms-for-initsite` returns a *wfdata* container (symbol *wfdata-con*), which have been checked for quality. The *wfdata* objects are specifically generated for the origin, station, and time interval required to perform beamforming in support of noise amplitude estimation. If *wfdata-con* proves to be empty, a warning message is returned, processing stops in this function, and control is returned to the calling program. This function acquires waveforms by a variation of the standard method in *DFX*. For further details see “Reading Waveforms” on page 58.

### **Estimate Origin Amplitudes for Predicted Arrivals [7.5.5]**

This subprocess measures amplitudes on the waveforms. It is implemented by the global Scheme function `measure-origin-amplitudes` and is called by Scheme language code such as:

## ▼ Seismic Processing

```
(set! dbamplitude-dbparrival-list
  (measure-origin-amplitudes wfdata-con site origin
    amprec-con))
```

The function requires four arguments, which are described in Table 121.

**TABLE 121: FUNCTIONAL INTERFACE OF MEASURE-ORIGIN-AMPLITUDES**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>site</i>	input	GObj	<i>dbsite</i> object
<i>origin</i>	input	GObj	<i>dborigin</i> object
<i>amprec-con</i>	input	container	<i>amplitude recipe</i> objects
n/a	returned value	list	<i>dbamplitude</i> and <i>dbparrival</i> containers

Given a *wfdata* container, a *dbsite* object, a *dborigin* object, and a container of amplitude estimation recipes, *measure-origin-amplitudes* estimates the origin-based amplitudes by calling the global Scheme function *make-parrival-amplitudes*. The function *measure-origin-amplitudes* creates and returns a list consisting of a *dbamplitude* container and a *dbparrival* container, each of which contains an object for each amplitude estimation recipe. The global Scheme function *make-parrival-amplitudes* calls the fundamental *DFX/libamp* function *measure-origin-amplitude*, which performs the amplitude measurements. See [IDC5.2.1] for more information about estimating waveform amplitudes for seismic processing.

The returned list (symbol *dbamplitude-dbparrival-list*) of a *dbamplitude* container and a *dbparrival* container is decomposed into these two containers by the inline generic Scheme functions *car* and *cadr*. If either container is empty, a warning message is returned, processing stops in this function, and control is returned to the calling program. If neither container is empty processing continues.

### Write Estimates for Predicted Arrivals [7.5.6]

This subprocess is implemented by the local Scheme function `submit-aux`. It is called by the Scheme language code:

```
(submit-aux dbresults-con results-table)
```

The function `submit-aux` requires two arguments, symbols *dbresults-con* and *results-table*, which are described in Table 74 on page 163. Symbols *dbYcon* and *table\_name* in that table correspond to symbols *dbresults-con* and *results-table* here.

The function `submit-aux` submits the container of results (symbol *dbresults-con*) to the database for writing into the appropriate table (symbol *results-table*) via the global Scheme function `submit-container-db`. This function returns `t` if successful, and `nil` otherwise. There are two successive calls to `submit-aux`: the first is for the *dbamplitude* container contents that are written to the **amplitude** database table, and the second is for the *dbparrival* container contents that are written to the **parrival** database table. If submission to the database is unsuccessful:

1. The database is rolled back to the state it had at the time of the last commit.
2. A warning message is printed.
3. Processing stops in this function.
4. Control is returned to the function `process-origins`.
5. Processing proceeds with the next station.

If both database submissions are successful processing continues. Following the completion of all six subprocesses, the database entries are committed by the global Scheme function `commit-db`. The local waveform container and its associated masks are then explicitly subject to garbage collection by the global Scheme function `gc-wfdata-con`. This latter function is called by the Scheme language code:

```
(gc-wfdata-con wfdata-con)
```

The function requires one argument, which is described in Table 122.

## ▼ Seismic Processing

TABLE 122: FUNCTIONAL INTERFACE OF GC-WFDATA-CON

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
n/a	returned value	logical scalar	indicates success ( <i>t</i> ) or failure ( <i>nil</i> )

The function `gc-wfdata-con` returns `t` if successful and `nil` otherwise.

## AUTOMATIC SEISMIC EVENT CHARACTERIZATION [8]

*Automatic Seismic Event Characterization (SeisEvch)* is a DFX Scheme application that extracts signal features, which can be used to characterize the nature of the source of the event. The source code resides in the file `DFX-evch.scm`. *SeisEvch* is usually run by specifying the network, start-time, end-time, and the `DFX-evch.par` file on the command line. It is generally executed by *tuxshell* within the Event Characterization pipeline. The results of *SeisEvch* are written to the following database tables: **amplitude**, **ceppks**, **complexity**, **parrival**, **splp**, **spvar**, **timefreq**, and **thirdmom**.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Estimate Event Characteristics* [8.3]

### General Description

*SeisEvch* processing was designed to make measurements on waveforms associated with events. The measurements may be used in event screening to identify particular events as having natural underlying causes. *SeisEvch* is intended to operate on a set of origins defined by a time interval and a set of stations defined by a seismic sensor network specified at run-time. Given the network and the processing time interval, the queries to the database yield a set of events formed during the time interval and a set of site information affiliated with the network.

## ▼ Seismic Processing

The time intervals to use for feature estimation at each station are well defined for origin-based processing. This issue is very important because in passive signal processing in general, and for estimation specifically, the quality of feature information extracted from a detected signal is often quite sensitive to the time alignment of the processing window within which the feature estimation is performed.

This application extracts a variety of signal attributes by computing a diversity of estimates of the waveform properties in both the time and frequency domains. Specifically, (i) several amplitude measures are extracted from the detected signal as well as for theoretical arrivals (that is, predicted arrivals where no detections are made); (ii) comparisons are made between short-period and long-period energy levels; (iii) source-multiplicity features are computed, and cepstral analysis is performed; (iv) spectral variance and time-frequency measures are made; and (v) statistical attributes are estimated, for example, the third-moment of frequency. These attributes are believed to support characterization of some events as having arisen from natural causes. The estimates made are saved in appropriate database tables for subsequent analysis.

**Architecture**

Figure 32 shows the top-level architecture of *SeisEvch*. This application is implemented by a single pass through a sequence of two initialization processes, followed by a nested pair of loops within which the principal computational processing is accomplished. The initialization processes acquire the relevant origin information from the database and the site-specific information for all stations relevant to the origins. The main processing occurs within a nested pair of loops. The outer one loops over origins, and the inner one loops over stations. The signal processing and feature estimation is contained within the inner loop and is implemented by the process that estimates event characteristics.

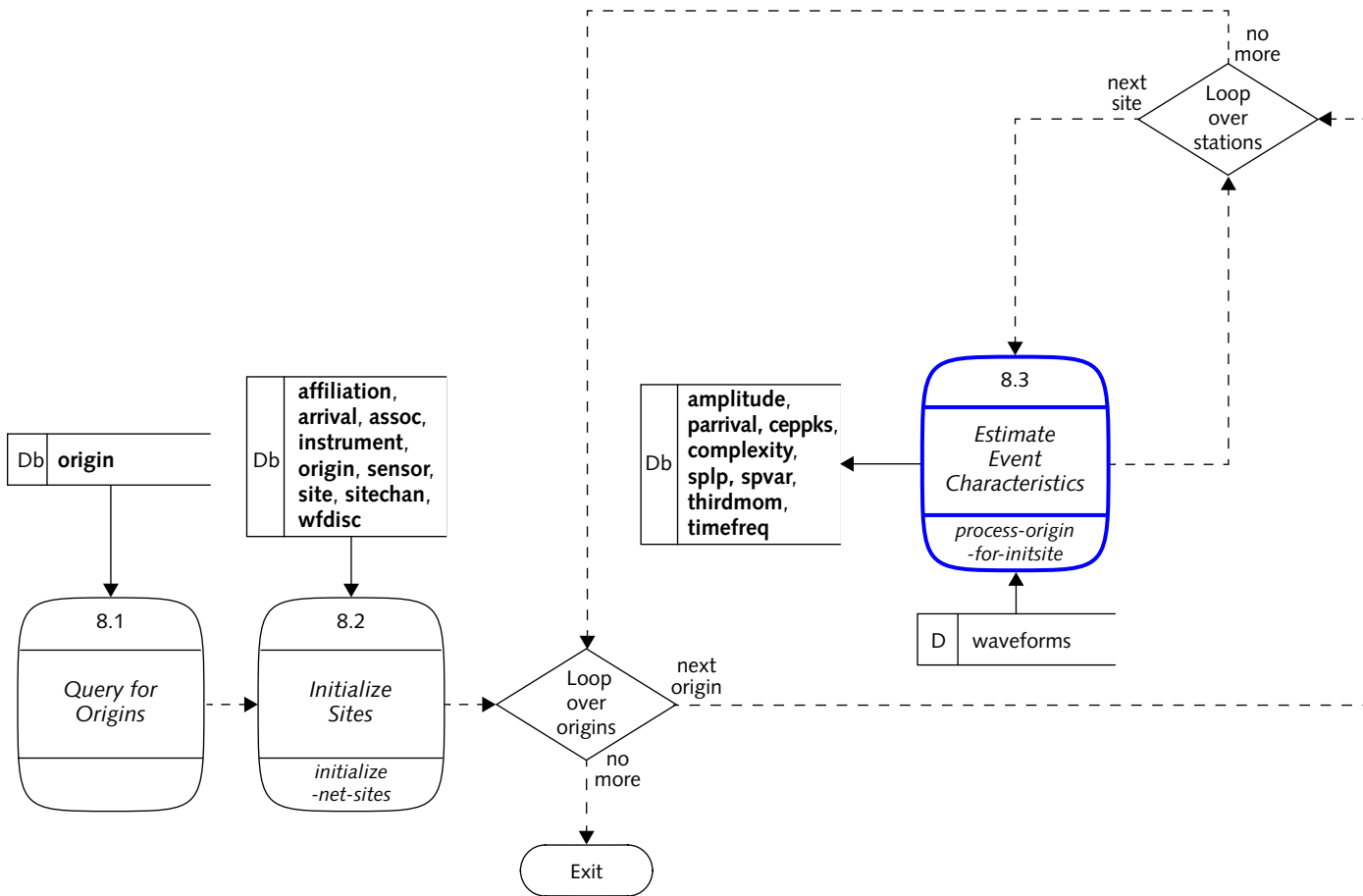


FIGURE 32. ARCHITECTURE OF SEISEVCH

## ▼ Seismic Processing

**Input/Output**

*SeisEvch* is called for a given network and time interval. The network code, start-time, and end-time parameters are generally specified on the command line. The rest of the parameters are set or referenced in the `DFX-evch.par` file. The input CVAR datastore parameters used in the Scheme code are described in Table 123.

**TABLE 123: INPUT PARAMETERS FOR SEISMIC EVENT CHARACTERIZATION PROCESSING**

Par Name	Source	Default Value	Description
<i>net</i>	command line	<none>	network of stations to process
<i>start-time</i>	command line	<none>	start-time of the processing interval
<i>end-time</i>	command line	<none>	end-time of the processing interval
<i>database-account</i>	par file	<none>	database access character-string
<i>NetType</i>	par file	<none>	network type, as in: "ss", "array", "larray"
<i>StaType</i>	par file	<none>	station type, as in: "1c", "3c"
<i>BandType1</i>	par file	<none>	standard frequency band for single component sensors
<i>BandType3</i>	par file	<none>	standard frequency band for 3-C sensors
<i>wfdisc-extension-len</i>	par file	86400.0	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX file-system



**TABLE 123: INPUT PARAMETERS FOR SEISMIC EVENT CHARACTERIZATION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>data-output-wftag</i>	par file	0	flag for <b>wftag</b> table output: 0 means do not write to <b>wftag</b> table
<i>evch-data-offset-len</i>	par file	<none>	time duration of extra waveforms at the start and end of the processing interval
<i>evch-data-tirec-list</i>	par file	<none>	list of time-interval recipes
<i>evch-site-recipe-file</i>	par file	<none>	name of the “.par” file containing site-specific event characterization processing recipes
<i>evch-arrivalamp-amprec-list</i>	par file	<none>	list of processing recipes to estimate arrival amplitudes
<i>evch-arrivalamp-max-depth</i>	par file	<none>	maximum depth of events to be processed for arrival amplitude
<i>evch-arrivalamp-min-delta</i>	par file	<none>	minimum distance of events to be processed for arrival amplitude (degrees)
<i>evch-arrivalamp-max-delta</i>	par file	<none>	maximum distance of events to be processed for arrival amplitude (degrees)
<i>evch-complexity-phase-list</i>	par file	<none>	list of phases to be used in the complexity processing
<i>evch-complexity-beamrec</i>	par file	<none>	beam-recipe to be used in support of the complexity analysis

## ▼ Seismic Processing

**TABLE 123: INPUT PARAMETERS FOR SEISMIC EVENT CHARACTERIZATION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>evch-complexity-min-delta</i>	par file	<none>	minimum distance of events to be processed for complexity (degrees)
<i>evch-complexity-max-delta</i>	par file	<none>	maximum distance of events to be processed for complexity (degrees)
<i>evch-fm-amprec-list</i>	par file	<none>	list of amplitude processing recipes to be used in first motion analysis
<i>evch-fm-phase-list</i>	par file	<none>	list of phases to be used in first motion analysis
<i>evch-originamp-amprec-list</i>	par file	<none>	list of processing recipes to estimate origin amplitudes
<i>evch-originamp-max-depth</i>	par file	<none>	maximum depth of events to be processed for origin amplitude
<i>evch-originamp-min-delta</i>	par file	<none>	minimum distance of events to be processed for origin amplitude (degrees)
<i>evch-originamp-max-delta</i>	par file	<none>	maximum distance of events to be processed for origin amplitude (degrees)
<i>evch-PcP-phase-list</i>	par file	<none>	list of PcP phases
<i>evch-PcP-min-delta</i>	par file	<none>	minimum distance of PcP phase (degrees)
<i>evch-PcP-max-delta</i>	par file	<none>	maximum distance of PcP phase (degrees)

**TABLE 123: INPUT PARAMETERS FOR SEISMIC EVENT CHARACTERIZATION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>evch-smult-max-depth</i>	par file	<none>	maximum depth of events to be processed for source multiplicity
<i>evch-smult-min-delta</i>	par file	<none>	minimum distance of events to be processed for source multiplicity (degrees)
<i>evch-smult-max-delta</i>	par file	<none>	maximum distance of events to be processed for source multiplicity (degrees)
<i>evch-splp-max-depth</i>	par file	<none>	maximum depth of events to be processed for short-period-versus-long-period analysis
<i>evch-splp-min-delta</i>	par file	<none>	minimum distance of events to be processed for short-period-versus-long-period analysis (degrees)
<i>evch-splp-max-delta</i>	par file	<none>	maximum distance of events to be processed for short-period-versus-long-period analysis
<i>evch-tf-phase-list</i>	par file	<none>	list of phases to be used in the time-frequency analysis
<i>evch-tf-max-depth</i>	par file	<none>	maximum depth of events to be processed for time-frequency analysis
<i>evch-tf-min-delta</i>	par file	<none>	minimum distance of events to be processed for time-frequency analysis (degrees)

## ▼ Seismic Processing

**TABLE 123: INPUT PARAMETERS FOR SEISMIC EVENT CHARACTERIZATION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>evch-tf-max-delta</i>	par file	<none>	maximum distance of events to be processed for time-frequency analysis (degrees)
<i>evch-tmf-phase-list</i>	par file	<none>	list of phases to be used in the third-moment-of-frequency analysis
<i>evch-tmf-min-delta</i>	par file	<none>	minimum distance of events to be processed for third-moment-of-frequency analysis (degrees)
<i>evch-tmf-max-delta</i>	par file	<none>	maximum distance of events to be processed for third-moment-of-frequency analysis (degrees)
<i>tmf-fmin</i>	par file	<none>	minimum frequency to be used for third-moment-of-frequency analysis
<i>tmf-fmax</i>	par file	<none>	maximum frequency to be used for third-moment-of-frequency analysis
<i>affiliation-table</i>	par file	affiliation	table containing station and network affiliations
<i>instrument-table</i>	par file	instrument	table containing generic (default) calibration information about a station
<i>origin-table</i>	par file	origin	table containing origin information

**TABLE 123: INPUT PARAMETERS FOR SEISMIC EVENT CHARACTERIZATION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>sensor-table</i>	par file	sensor	table containing specific calibration information for physical channels
<i>site-table</i>	par file	site	table containing site-location information
<i>sitechan-table</i>	par file	sitechan	table containing station-channel information
<i>in-arrival-table</i>	par file	arrival	input table containing arrival information
<i>in-assoc-table</i>	par file	assoc	input table containing association information
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information
<i>origerr-table</i>	par file	origerr	database table containing summary of errors in origin estimation
<i>lastid-table</i>	par file	lastid	database table containing current maximum allocated values of identification numbers for various tables
<i>out-amplitude-table</i>	par file	amplitude	output table containing amplitude information generated by this application
<i>out-ceppks-table</i>	par file	ceppks	output table containing cepstral peaks information generated by this application

## ▼ Seismic Processing

**TABLE 123: INPUT PARAMETERS FOR SEISMIC EVENT CHARACTERIZATION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>out-complexity-table</i>	par file	complexity	output table containing complexity information generated by this application
<i>out-parrival-table</i>	par file	parrival	output table containing predicted arrival information generated by this application
<i>out-splp-table</i>	par file	splp	output table containing short-period-versus-long-period information generated by this application
<i>out-spvar-table</i>	par file	spvar	output table containing spectral variance information generated by this application
<i>out-tf-table</i>	par file	timefreq	output table containing time-frequency information generated by this application
<i>out-tmf-table</i>	par file	thirdmom	output table containing third-moment-of-frequency information generated by this application

Table 124 lists the output of *SeisEvch*. The process writes the results to the database, but first checks the database for results that already exist by comparing arrival times, arids, orids, or other such indicative information. Those results that do not already exist are written to the appropriate database tables. This processing, which avoids duplication of results in the database, follows the conventions described in the “Screening Database Containers” on page 43.

**TABLE 124: DATA PRODUCED BY SEISMIC EVENT CHARACTERIZATION PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>amplitude</b>	database table	waveform amplitude attributes
<b>ceppks</b>	database table	cepstral analysis attributes
<b>complexity</b>	database table	complexity analysis attributes
<b>parrival</b>	database table	predicted arrival attributes
<b>splp</b>	database table	short-period-versus-long-period attributes
<b>spvar</b>	database table	spectral variance attributes
<b>timefreq</b>	database table	time-frequency analysis attributes
<b>thirdmom</b>	database table	third-moment-of-frequency attributes

Table 237 on page 448 indicates the specific attributes written for each of these database tables.

## Processes

The following processes shown in Figure 32 on page 251 are described in this section:

- *Query for Origins [8.1]*
- *Initialize Sites [8.2]*
- *Estimate Event Characteristics [8.3]*

### Query for Origins [8.1]

Because *SeisEvch* bases its processing on origins, the first important information it obtains is the set of origins for the given processing time interval. Origin information includes the event's time, geographic location, and depth, as well as a variety

## ▼ Seismic Processing

of magnitude measures of the event. The process *Query for Origins* acquires information about the relevant origins. Processing begins with the acquisition of three parameters from the CVAR datastore: the values for *start-time* and *end-time* for this processing time interval and the name of the origin table in the database. Processing then continues with a query to the database for origins during the time span of this interval via the local Scheme function `query-for-dborigin-by-time`. For general information on how database queries are implemented see “Querying the Database” on page 41.

The function `query-for-dborigin-by-time` is called by Scheme language code such as:

```
(set! dborigin-con (query-for-dborigin-by-time start-time
end-time origin-table))
```

The function requires three arguments, which are described in Table 125.

**TABLE 125: FUNCTIONAL INTERFACE OF QUERY-FOR-DBORIGIN-BY-TIME**

Argument Name	Usage	Type	Description
<i>start-time</i>	input	real	start-time of the processing interval
<i>end-time</i>	input	real	end-time of the processing interval
<i>dborigin-table</i>	input	character-string	<b>origin</b> table name
n/a	returned value	container	<i>dborigin</i> objects

The function `query-for-dborigin-by-time` assumes at the outset that the database is opened. It constructs the query string in accordance with the standard procedures. The query string is submitted to the database by the global Scheme function `query-for-container`. The returned entity is a *dborigin* container. If no origins are found a message is printed and the application exits.



### Initialize Sites [8.2]

This process acquires the site-specific information required by *SeisEvch* and stores it in an *initsite* container. The process is implemented by the local Scheme function `initialize-net-sites`. This function is called by Scheme language code such as:

```
(set! initsite-con (initialize-net-sites dborigin-con))
```

The function requires one argument, which is described in Table 68 on page 155.

This function accesses the CVAR datastore as well as the contents of the *dborigin* container to set the values of the internal symbols described in Table 126.

**TABLE 126: INTERNAL SYMBOLS FOR INITIALIZE-NET-SITES**

Symbol	Type	Source	Description
<i>net</i>	character-string	CVAR	network of stations to process
<i>origin-time-list</i>	list	<i>dborigin-con</i>	real-valued origin times
<i>start-time</i>	real	<i>dborigin-con</i>	minimum time in the <i>origin-time-list</i>
<i>end-time</i>	real	<i>dborigin-con</i>	end-time of the processing interval maximum time in the <i>origin-time-list</i>
<i>orid-list</i>	list	<i>dborigin-con</i>	integers: unique origin identification numbers
<i>ex-time</i>	real	CVAR	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>arrival-table</i>	character-string	CVAR	<b>arrival</b> table name
<i>assoc-table</i>	character-string	CVAR	<b>assoc</b> table name
<i>origin-table</i>	character-string	CVAR	<b>origin</b> table name

## ▼ Seismic Processing

TABLE 126: INTERNAL SYMBOLS FOR INITIALIZE-NET-SITES (CONTINUED)

Symbol	Type	Source	Description
<i>wfdisc-table</i>	character-string	CVAR	<b>wfdisc</b> table name
<i>affiliation-table</i>	character-string	CVAR	table containing station and network affiliations
<i>instrument-table</i>	character-string	CVAR	table containing generic (default) calibration information about a station
<i>sensor-table</i>	character-string	CVAR	table containing specific calibration information for physical channels
<i>site-table</i>	character-string	CVAR	table containing site-location information
<i>sitechan-table</i>	character-string	CVAR	table containing station-channel information

The process *Initialize Sites* is a special case of the Initialize Site function (see “Initializing Sites” on page 46). In addition to the standard activities described therein, the function `initialize-net-sites` performs the following steps:

1. Initialize the travel-time tables via the local Scheme function `initialize-net-tt-tables`.
2. Calculate the waveform intervals via the local Scheme function `compute-net-ti`.
3. Bind the symbols described in Table 127 to non-nil values, strings, lists, objects, and containers via a set of queries to the relevant database tables. For further information on how database queries are implemented see “Querying the Database” on page 41.

**TABLE 127: INTERNAL SYMBOLS SET FROM DATABASE QUERIES WITHIN INITIALIZE-NET-SITES**

Symbol	Type	Source	Description
<i>dbarsoc-con</i>	container	<b>affiliation</b> table <b>arrival</b> table <b>assoc</b> table <b>origin</b> table	<i>dbarsoc</i> objects: associated arrivals for the given network stations
<i>sta-list</i>	list	<i>dbarsoc-con</i>	character-strings: unique station codes for the given <i>dbarsocs</i>
<i>dbsite-con</i>	container	<b>affiliation</b> table <b>site</b> table	<i>dbsite</i> container: stations for the associated arrivals
<i>wfdata-con</i>	container	<b>affiliation</b> table <b>instrument</b> table <b>sensor</b> table <b>site</b> table <b>sitechan</b> table	<i>wfdata</i> container: for the time interval, station list and channel list relevant to the origins in this processing time interval (extended by ex-time)
<i>wfmem-con</i>	container	<b>affiliation</b> table <b>wfdisc</b> table	<i>wfmem</i> container: for the time interval, station list and channel list relevant to the origins in this processing time interval (extended by ex-time)

This function returns an *initsite* container consisting of members described in the eighth column of Table 8 on page 50. If any of the containers described in Table 8 are returned empty, an error message is returned, and the application exits.

After the two processes *Query for Origins* and *Initialize Sites* are complete, the computational processing is accomplished in an architecture that is implemented with a pair of nested loops. The outer loop runs over origins, and for each given origin the inner loop runs over the stations. (That is, the inner loop is over all stations relevant to that origin.) The processing for a given origin terminates when there are no further stations (*initsite* objects) to be evaluated, and the processing

## ▼ Seismic Processing

for a given time interval terminates when there are no further origins (*dborigin* objects) for that time interval. The computational processing interior to the nested loops is implemented by the process *Estimate Event Characteristics*, which is called by the local Scheme function `process-origins-for-init-site`. The processing within this function writes all of the output to the database that results from the computations within *SeisEvch*.

### Estimate Event Characteristics [8.3]

This process acquires the waveforms and computes the values of the signal and waveform attributes. It also saves the results of these computations in the appropriate database tables. The process is implemented by the local Scheme function `process-origin-for-init-site`. This function is called by Scheme language code such as:

```
(process-origin-for-init-site init-site dborigin dborigerr)
```

The function requires three arguments, which are described in Table 128.

**TABLE 128: FUNCTIONAL INTERFACE OF QUERY-FOR-DBORIGIN-BY-TIME**

Argument Name	Usage	Type	Description
<i>init-site</i>	input	GObj	<i>init-site</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>dborigerr</i>	input	GObj	<i>dborigerr</i> object
n/a	returned value	scalar	indicates success (t) or failure (nil)

This function operates on a single origin and station. For each single origin and for a given station associated with that origin, the function `process-origin-for-init-site` performs the following steps shown in Figure 33:

1. Initialize processing via the subprocess *Extract Parameters*.
2. Acquire waveforms via the subprocess *Acquire Waveforms*.

3. Estimate the theoretical amplitudes via the subprocess *Perform Theoretical Amplitude Processing*.
4. Estimate the observed amplitudes via the subprocess *Perform Observed Amplitude Processing*.
5. Estimate the first motion amplitudes via the subprocess *Perform First Motion Amplitude Processing*.
6. Estimate time-frequency measures via the subprocess *Perform Time-frequency Processing*.
7. Estimate the third-moment-of-frequency measures via the subprocess *Perform Third-moment-of-frequency Processing*.
8. Estimate the short-period-versus-long-period measures via the subprocess *Perform Short-period-versus-Long-period Processing*.
9. Estimate the source-multiplicity measures via the subprocess *Perform Source Multiplicity Processing*.
10. Estimate the complexity measures via the subprocess *Perform Complexity Processing*.

See [IDC5.2.1] for more information on the various attributes referenced in these steps as well as the algorithms with which they are estimated.

Warning messages are printed for any feature computation failures and existing measurement screening. This can occur for the subprocesses in steps 3 through 10.

Any of the following conditions cause an error message to be printed and processing to stop:

- either of the input arguments *initsite* or *dborigin* is null
- the *dbarsoc* container within the *initsite* object contains no associated arrivals for the current station
- the *wfdata* container is null

For more information about this function see the following section, Detailed Description of Estimate Event Characteristics [8.3].

## ▼ Seismic Processing

**Detailed Description of Estimate  
Event Characteristics [8.3]**

This process acquires the waveforms and estimates the signal features used for event characterization. It is implemented by a call to the local Scheme function `process-origin-for-initsite`.

Figure 33 shows the architecture of this process, which consists of a single pass through ten subprocesses. The first two subprocesses extract parameters from the *initsite* object and *dborigin* object and acquire waveforms for this station for the time interval relevant to this origin. The remaining subprocesses perform theoretical amplitude processing, observed amplitude processing, first-motion amplitude processing, time-frequency (tf) processing, third-moment-of-frequency (tmf) processing, short-period-versus-long-period (splp) processing, source-multiplicity (including cepstral analysis) processing, and finally complexity processing. For subprocesses 8.3.3 through 8.3.10, as shown in Figure 33, certain criteria must be satisfied for the feature to be estimated. Table 129 on page 268 lists the criteria for each feature.

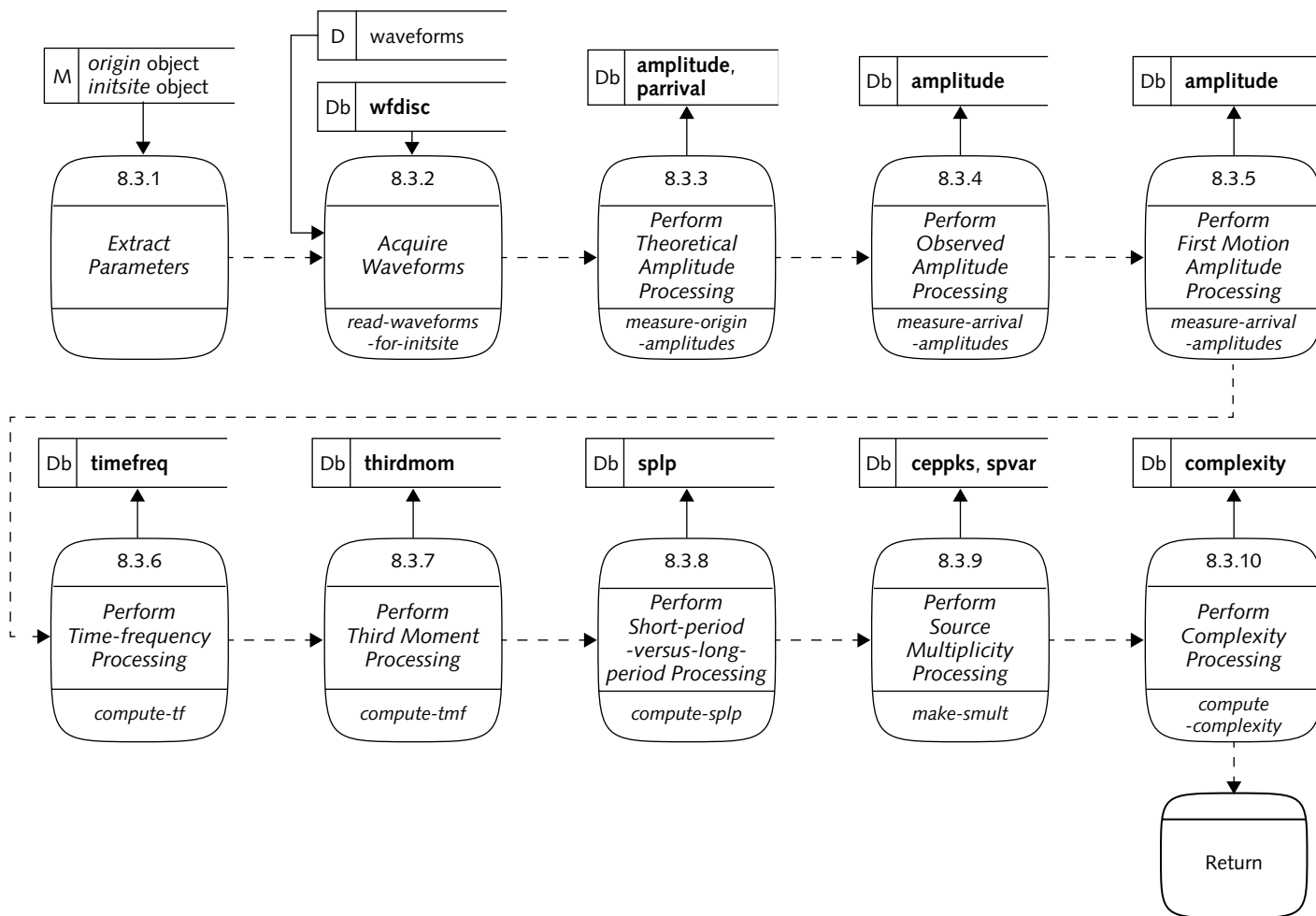


FIGURE 33. ARCHITECTURE OF ESTIMATE EVENT CHARACTERISTICS

## ▼ Seismic Processing

TABLE 129: CRITERIA TO BE SATISFIED FOR FEATURE ESTIMATION

Feature	Criteria	Parameters Governing Acceptance
theoretical amplitude	distance: station-to-event	limit: <i>evch-originamp-min-delta</i> limit: <i>evch-originamp-max-delta</i>
	depth: event	limit: <i>evch-originamp-max-depth</i>
observed amplitude	distance: station-to-event	limit: <i>evch-arrivalamp-min-delta</i> limit: <i>evch-arrivalamp-max-delta</i>
	depth: event	limit: <i>evch-arrivalamp-max-depth</i>
first-motion amplitude	phase	contained in: <i>evch-fm-phase-list</i>
time-frequency	distance: station-to-event	limit: <i>evch-tf-min-delta</i> limit: <i>evch-tf-max-delta</i>
	depth: event	limit: <i>evch-tf-max-depth</i>
third-moment-of-frequency	distance: station-to-event	limit: <i>evch-tmf-min-delta</i> limit: <i>evch-tmf-max-delta</i>
	phase	contained in: <i>evch-tmf-phase-list</i>
short-period-versus-long-period	distance: station-to-event	limit: <i>evch-splp-min-delta</i> limit: <i>evch-splp-max-delta</i>
	depth: event	limit: <i>evch-splp-max-depth</i>
source multiplicity	distance: station-to-event	limit: <i>evch-smult-min-delta</i> limit: <i>evch-smult-max-delta</i>
	depth: event	limit: <i>evch-smult-max-depth</i>
complexity	distance: station-to-event	limit: <i>evch-complexity-min-delta</i> limit: <i>evch-complexity-max-delta</i> AND NOT
	distance: station-to-event	limit: <i>evch-PcP-min-delta</i> limit: <i>evch-PcP-max-delta</i> AND
	phase	contained in: <i>evch-PcP-list</i>



At the conclusion of this sequence of subprocesses, the process commits the database, cleans up memory associated with waveform sets, and returns control to the calling program.

Prior to the principal processing within the function `process-origin-for-init-site` the following steps are performed:

1. A local Scheme function, `submit-aux`, is defined.
2. Memory clean-up is explicitly ensured by invoking the global Scheme function `gc` (garbage collection).
3. The input arguments are checked to ensure that neither the *init-site* nor the *db-origin* input argument is `nil`. If either argument is `nil`, an error message is printed and processing continues with the next station.

The function `submit-aux` submits results to the database via customary use of the global Scheme function `submit-container-db` and, where the submission is successful, it returns `t`. If the submission fails, then the database is rolled back via the global Scheme function `rollback-db` to the state it had before the attempted submission. In this case of failed submission `submit-aux` returns `nil`.

The following subprocesses shown in Figure 33 on page 267 are described in this section:

- Extract Parameters [8.3.1]
- Acquire Waveforms [8.3.2]
- Perform Theoretical Amplitude Processing [8.3.3]
- Perform Observed Amplitude Processing [8.3.4]
- Perform First Motion Amplitude Processing [8.3.5]
- Perform Time-frequency Processing [8.3.6]
- Perform Third-moment-of-frequency Processing [8.3.7]
- Perform Short-period-versus-long-period Processing [8.3.8]
- Perform Source-multiplicity Processing [8.3.9]
- Perform Complexity Processing [8.3.10]

## ▼ Seismic Processing

**Extract Parameters [8.3.1]**

This subprocess acquires from the *initsite* and *dborigin* objects all of the parameters required to perform the attribute computations that follow. In addition, this subprocess obtains an *arsoc* container whose contents are associated with the origin being processed and the station codes corresponding to the *initsite*. If the *arsocs* container is empty, an error message is printed, and processing continues with the next station. Then, the local symbol *delta* is set to the value of the "*delta*" parameter (station-to-event distance) in the first *arsoc* object in the extracted subset.

**Acquire Waveforms [8.3.2]**

The process *Estimate Event Characteristics* uses waveforms to estimate the attributes of each detected signal. These data are acquired for an appropriate time-interval for each of the stations relevant to the event being processed. The time-interval is calculated to ensure that the waveforms span the detected signal at the observed or theoretical arrival time. The subprocess *Acquire Waveforms* accomplishes this requirement. It is implemented by the local Scheme function *read-waveforms-for-initsite*. This function is called by Scheme language code such as:

```
(set! wfdata-con (read-waveforms-for-initsite initsite
dborigin))
```

The function requires two arguments, which are described in Table 130.

**TABLE 130: FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-INITSITE**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	<i>initsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
n/a	returned value	container	<i>wfdata</i> objects

This function returns a *wfdata* container (symbol *wfdata-con*) that contains the waveforms for the station being processed using a time interval predicted by the current origin. This function acquires waveforms by a variation of the standard method in *DFX* (see “Reading Waveforms” on page 58).

The function `read-waveforms-for-initsite` performs the following steps:

1. Make a local copy of the *wfdata* container within the *initsite* object.
2. Extract the appropriate subset of time-interval recipes, and determine the time interval for processing via the *DFX/libutil* function `compute-site-origin-ti`. The placement and extent of the time interval is based on the station-origin distance. Adjustments to the time interval are made in accordance with *evch-data-offset-len* to avoid edge effects in beam generation intervals.
3. Fill in the *wfmem* container as well as the *wfmem* objects within the *wfdata* objects via the *DFX/libdata* function `read-waveforms`.
4. Check the quality of the waveforms via the *DFX/libqc* function `qc-waveforms` using the *quality-control recipe* object contained in the *initsite* object.
5. Discard stations/channels from the *wfdata* container identified by quality-control processing as too poor in quality. If the *wfdata* container is empty following the quality-control processing, an error message is printed and processing continues with the next station.

### **Perform Theoretical Amplitude Processing [8.3.3]**

This subprocess produces amplitude estimates from the noise present on stations that did not yield an associated detection. Only origins that fall within a minimum and maximum distance range are processed. The station-to-event distance criteria are set by the parameters *evch-originamp-min-delta* and *evch-originamp-max-delta*. In addition, the origin must be shallower than a depth threshold set by the symbol *evch-originamp-max-depth*, taking into account the uncertainty in the depth estimate. A set of amplitude-estimation processing recipes from the *initsite* object is

## ▼ Seismic Processing

used to compute the theoretical amplitudes. The time interval during which the amplitude is estimated is based upon a prediction for the theoretical travel time given the station-to-event distance and the phase. This subprocess is performed by the global Scheme function `measure-origin-amplitudes`, which is called by the Scheme language code:

```
(set! dbamplitude-dbparrrival-list
  (measure-origin-amplitudes wfdata-con dbsite dborigin
    theo-amprec-con))
```

The function requires four arguments, which are described in Table 131.

**TABLE 131: FUNCTIONAL INTERFACE OF MEASURE-ORIGIN-AMPLITUDES**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>theo-amprec-con</i>	input	container	<i>amplitude estimation processing recipe</i> objects
n/a	returned value	list	<i>dbamplitude</i> and <i>dbparrrival</i> containers

The global Scheme function `measure-origin-amplitudes` is a wrapper to the *DFX/libamp* function `measure-origin-amplitude` (note the singular form), which actually performs the amplitude estimation processing.

The function `measure-origin-amplitudes` returns a list of *dbamplitude* and *dbparrrival* containers. If this list is nil a warning message is printed. Otherwise, the newly generated *dbparrrival* and *dbamplitude* datasets are compared with *parrival* or *amplitude* records that already exist in the database. A duplicate *parrival* is determined by matching the *evid*, *orid*, *sta*, and *phase* attributes. For *dbamplitudes* the matching attributes are *chan* and *amptype*. Any duplicates found in the new datasets are removed from the appropriate container. This subprocess follows the general form described in “Screening Database Containers” on page 43. The

screening function is accomplished by the global Scheme function `screen-origin-dbamplitude-by-existing-dbamplitude`. This function is called by the Scheme language code:

```
(screen-origin-dbamplitude-by-existing-dbamplitude
 dbparrival-con dbamplitude-con)
```

The function requires two argument, which are described in Table 132.

**TABLE 132: FUNCTIONAL INTERFACE OF SCREEN-ORIGIN-DBAMPLITUDE-BY-EXISTING-DBAMPLITUDE**

Argument Name	Usage	Type	Description
<i>dbparrival-con</i>	input and output	container	<i>dbparrival</i> objects
<i>dbamplitude-con</i>	input and output	container	<i>dbamplitude</i> objects
n/a	returned value	list	<i>dbamplitude</i> and <i>dbparrival</i> containers

If all of the *dbamplitude* objects are removed in the screening process, an informational message is printed. Otherwise, the *dbamplitude* container is submitted to the database whereby the results are written to the **amplitude** table. Similarly, if all of the *dbparrival* objects are removed in the screening process a message is printed. Otherwise, the *dbparrival* container is submitted to the database whereby the results are written to the **parrival** table. These submission processes are implemented by the local Scheme function `submit-aux`, as previously discussed.

### Perform Observed Amplitude Processing [8.3.4]

This subprocess produces amplitude estimates from observed detections. Only origins that fall within a specified distance range are processed. The station-to-event distance criteria are set by the parameters *evch-arrivalamp-min-delta* and *evch-arrivalamp-max-delta*. In addition, the origin must be shallower than a maximum depth threshold set by the symbol *evch-arrivalamp-max-depth*, taking into account the uncertainty in the depth estimate. A set of amplitude estimation processing

## ▼ Seismic Processing

recipes from the *initsite* object is used to compute the observed amplitudes. This subprocess is performed by the global Scheme function `measure-arrival-amplitudes`, which is called by the Scheme language code:

```
(set! dbamplitude-con (measure-arrival-amplitudes
  wfdata-con dbarsoc-con dbsite dborigin obs-amprec-con))
```

The function requires five arguments, which are described in Table 133.

**TABLE 133: FUNCTIONAL INTERFACE OF MEASURE-ARRIVAL-AMPLITUDES FOR OBSERVED ARRIVALS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>dbarsoc-con</i>	input	container	<i>dbarsoc</i> objects
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>obs-amprec-con</i>	input	container	<i>amplitude estimation processing recipe</i> objects
n/a	returned value	container	<i>dbamplitude</i> objects

The global Scheme function `measure-arrival-amplitudes` is a wrapper to the global Scheme `make-arrival-amplitudes`, which is itself a wrapper to the *DFX/libamp* function `measure-arrival-amplitude`. This function actually performs the amplitude estimation processing.

The function `measure-arrival-amplitudes` returns a *dbamplitude* container. If this container is empty a warning message is printed. Otherwise, processing continues with the calling of the global Scheme function `screen-arrival-dbamplitude-by-existing`. The newly generated *dbamplitude* datasets are compared with amplitude records that already exist in the database. A duplicate *dbamplitude* is determined by matching the *arid*, *chan*, and *amptype* attributes. Any duplicates found in the new datasets are removed from the container. This subprocess follows the general form described in the “Screening Database Containers”

on page 43. The screening function is implemented by the global Scheme function `screen-arrival-dbamplitude-by-existing`. This function is called by the Scheme language code:

```
(screen-arrival-dbamplitude-by-existing dbamplitude-con)
```

The function requires one argument, which is described in Table 134.

**TABLE 134: FUNCTIONAL INTERFACE OF SCREEN-ARRIVAL-DBAMPLITUDE-BY-EXISTING FOR OBSERVED ARRIVALS**

Argument Name	Usage	Type	Description
<i>dbamplitude-con</i>	input and output	container	<i>dbamplitude</i> objects
n/a	returned value	container	<i>dbamplitude</i> objects

If all of the *dbamplitude* objects are removed in the screening process a message is printed. Otherwise, the *dbamplitude* container is submitted to the database whereby the results are written to the **amplitude** table. This latter process is implemented by local Scheme function `submit-aux`, as previously discussed.

### Perform First Motion Amplitude Processing [8.3.5]

This subprocess produces amplitude estimates relevant to first motion. Only origins with phases matching one or more of those on the list of first motion phases is run through this subprocess. The list of first motion phases is set by the parameter *evch-fm-phase-list*. A set of amplitude estimation processing recipes from the *initsite* object is used to compute the first motion amplitudes. This subprocess is implemented by the global Scheme function `measure-arrival-amplitudes`, which is called by the Scheme language code:

```
(set! dbamplitude-con (measure-arrival-amplitudes
  wfdata-con fm-dbarsoc-con dbsite dborigin fm-amprec-con))
```

The function requires five arguments, which are described in Table 135.

## ▼ Seismic Processing

**TABLE 135: FUNCTIONAL INTERFACE OF MEASURE-ARRIVAL-AMPLITUDES FOR FIRST MOTION**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>fm-dbarsoc-con</i>	input	container	<i>dbarsoc</i> objects with phases suitable for first motion estimates
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>fm-amprec-con</i>	input	container	<i>amplitude estimation processing recipe</i> objects for first motion
n/a	returned value	container	<i>dbamplitude</i> objects

The global Scheme function `measure-arrival-amplitudes` is a wrapper to the global Scheme function `make-arrival-amplitudes`, which is itself a wrapper to the *DFX/libamp* function `measure-arrival-amplitude`. This function actually performs the amplitude estimation processing.

This function `measure-arrival-amplitudes` returns a *dbamplitude* container. If this container proves to be empty a warning message is printed. Otherwise, processing continues with the calling of the global Scheme function `screen-arrival-dbamplitude-by-existing`. The newly generated *dbamplitude* datasets are compared with amplitude records that already exist in the database. A duplicate *dbamplitude* is determined by matching the *arid*, *chan*, and *amptype* attributes. Any duplicates found in the new datasets are removed from the container. This subprocess follows the general form described in “Screening Database Containers” on page 43. The screening function is implemented by the global Scheme function `screen-arrival-dbamplitude-by-existing`. This function is called by the Scheme language code:

```
(screen-arrival-dbamplitude-by-existing dbamplitude-con)
```

The function requires one argument, which is described in Table 136.



**TABLE 136: FUNCTIONAL INTERFACE OF SCREEN-ARRIVAL-DBAMPLITUDE-BY-EXISTING FOR FIRST MOTION**

Argument Name	Usage	Type	Description
<i>dbamplitude-con</i>	input and output	container	<i>dbamplitude</i> objects
n/a	returned value	container	<i>dbamplitude</i> objects

If all of the *dbamplitude* objects are removed in the screening process a message is printed. Otherwise, the *dbamplitude* container is submitted to the database whereby the results are written to the **amplitude** table. This latter process is implemented by the local Scheme function `submit-aux`, as previously discussed.

### Perform Time-frequency Processing [8.3.6]

This subprocess produces time-frequency (tf) estimates for the detected signals. Only origins that fall within a specified distance range are processed. The station-to-event distance criteria are set by the parameters *evch-tf-min-delta* and *evch-tf-max-delta*. In addition, the origin must be shallower than a maximum depth criterion set by symbol *evch-tf-max-depth*. Finally, only origins with phases matching one or more of those on the list of time-frequency phases is run through this subprocess. The list of time-frequency phases is set by the parameter *evch-tf-phase-list*. A time-frequency estimation processing recipe from the *initsite* object is used to compute the time-frequency measures. This subprocess is performed by the local Scheme function `compute-tf`. This function is called by the Scheme language code:

```
(set! dbtf-con (compute-tf wfdata-con dborigin
  tf-dbarsoc-con tf-rec))
```

The function requires four arguments, which are described in Table 137.

## ▼ Seismic Processing

TABLE 137: FUNCTIONAL INTERFACE OF COMPUTE-TF

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>tf-dbarsoc-con</i>	input	container	<i>dbarsoc</i> objects for the arrival associations with phases appropriate to time-frequency analysis
<i>tf-rec</i>	input	GObj	<i>time-frequency estimation processing recipe</i> object
n/a	returned value	container	<i>dbtf</i> objects

The function `compute-tf` is a wrapper to the *DFX/libtf* function `cal-tf`, which actually performs the time-frequency analysis.

The function `compute-tf` returns a *dbtf* container. If this container is empty a warning message is printed. Otherwise, processing continues with the calling of the local Scheme function `screen-tf-by-existing-tf`. The newly generated *dbtf* datasets are compared with time-frequency records that already exist in the database. A duplicate *dbtf* is determined by matching the *orid* and *sta* attributes. Any duplicates found in the new datasets are removed from the container. This subprocess follows the general form described in the “Screening Database Containers” on page 43. The screening function is implemented by the global Scheme function `screen-tf-by-existing-tf`. This function is called by the Scheme language code:

```
(screen-tf-by-existing-tf dbtf-con)
```

The function requires one argument, which is described in Table 138.

TABLE 138: FUNCTIONAL INTERFACE OF SCREEN-TF-BY-EXISTING-TF

Argument Name	Usage	Type	Description
<i>dbtf-con</i>	input and output	container	<i>dbtf</i> objects
n/a	returned value	container	<i>dbtf</i> objects

If all of the *dbtf* objects are removed in the screening process a message is printed. Otherwise, the *dbtf* container is submitted to the database whereby the results are written to the **timefreq** table. This latter process is implemented by the local Scheme function `submit-aux`, as previously discussed.

### Perform Third-moment-of-frequency Processing [8.3.7]

This subprocess produces third-moment-of-frequency (tmf) estimates for the detected signals. Only origins that fall within a specified distance range are processed. The station-to-event distance criteria are set by the parameters *evch-tmf-min-delta* and *evch-tmf-max-delta*. In addition, only origins with phases matching one or more of those on the list of tmf phases is run through this subprocess. The list of tmf phases is set by the parameter *evch-tmf-phase-list*. A tmf estimation processing recipe from the *initsite* object is used to compute the tmf measures. This processing recipe is actually a locally modified version of the spectral multiplicity processing recipe. This subprocess is performed by the local Scheme function `compute-tmf`. This function is called by the Scheme language code:

```
(set! dbtmf-con (compute-tmf wfdata-con tmf-dbarsoc-con
  smult-rec))
```

The function requires three arguments, which are described in Table 139.

## ▼ Seismic Processing

TABLE 139: FUNCTIONAL INTERFACE OF COMPUTE-TMF

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>tmf-dbarsoc-con</i>	input	container	<i>dbarsoc</i> objects for the arrival associations with phases appropriate to tmf analysis
<i>smult-rec</i>	input	GObj	<i>smult estimation processing recipe</i> class with parameters modified to perform tmf calculations
n/a	returned value	container	<i>dbtmf</i> objects

The local Scheme function `compute-tmf` is a wrapper to the *DFX/libfs* function `make-tmf`, which actually performs the tmf analysis. Prior to execution of `compute-tmf` the *smult-rec* recipe object has its *smult-min-frequency* and *smult-max-frequency* attributes set to tmf-specific frequencies.

The function `compute-tmf` returns a *dbtmf* container. After the tmf analysis is completed, the *source-multiplicity recipe* object (symbol *smult-rec*) has its attributes *smult-min-frequency* and *smult-max-frequency* reset to their original values. If the *dbtmf* container proves to be empty a warning message is printed. Otherwise, processing continues with the calling of the local Scheme function `screen-tmf-by-existing-tmf`. The newly generated *dbtmf* datasets are compared with tmf records that already exist in the database (in the **thirdmom** table). A duplicate *dbtmf* is determined by matching the *arid* attribute. Any duplicates found in the new datasets are removed from the container. This subprocess follows the general form described in “Screening Database Containers” on page 43. The screening function is implemented by the global Scheme function `screen-tmf-by-existing-tmf`. This function is called by the Scheme language code:

```
(screen-tmf-by-existing-tmf dbtmf-con)
```

The function requires one argument, which is described in Table 140.

TABLE 140: FUNCTIONAL INTERFACE OF SCREEN-TMF-BY-EXISTING-TMF

Argument Name	Usage	Type	Description
<i>dbtmf-con</i>	input and output	container	<i>dbtmf</i> objects
n/a	returned value	container	GObjs of the <i>dbtmf</i> class

If all of the *dbtmf* objects are removed in the screening process a message is printed. Otherwise, the *dbtmf* container is submitted to the database whereby the results are written to the **thirdmom** table. This latter process is implemented by the local Scheme function `submit-aux`, as previously discussed.

#### Perform Short-period-versus-long-period Processing [8.3.8]

This subprocess produces short-period-versus-long-period (splp) estimates for the detected signals. Only origins that fall within a specified distance range are processed. The station-to-event distance criteria are set by the parameters *evch-splp-min-delta* and *evch-splp-max-delta*. In addition, the origin must be shallower than a maximum depth criterion set by the symbol *evch-splp-max-depth*. Finally, the seismic sensor must be a single-station 3-C instrument. A splp estimation processing recipe from the *initsite* object is used to compute the measure. This subprocess is implemented by the local Scheme function `compute-splp`. This function is called by the Scheme language code:

```
(set! dbsplp-con (compute-splp wfdata-con dborigin
                             dbarsoc-con splp-rec))
```

The function requires four arguments, which are described in Table 141.

## ▼ Seismic Processing

TABLE 141: FUNCTIONAL INTERFACE OF COMPUTE-SPLP

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>dbarsoc-con</i>	input	container	<i>dbarsoc</i> objects
<i>splp-rec</i>	input	container	<i>splp</i> estimation processing recipe objects
n/a	returned value	container	<i>dbsplp</i> objects

The local Scheme function `compute-splp` is a wrapper to the *DFX/libsplp* function `cal-splp`, which actually performs the *splp* estimation processing.

The function `compute-splp` returns a *dbsplp* container. If this container proves to be empty a warning message is printed. Otherwise, processing continues with the calling of the local Scheme function `screen-splp-by-existing-splp`. The newly generated *dbsplp* datasets are compared with *splp* records that already exist in the database. A duplicate *dbsplp* is determined by matching the *orid* and *sta* attributes. Any duplicates found in the new datasets are removed from the container. This subprocess follows the general form described in “Screening Database Containers” on page 43. This function is called by the Scheme language code:

```
(screen-splp-by-existing-splp dbsplp-con)
```

The function requires one argument, which is described in Table 142.

TABLE 142: FUNCTIONAL INTERFACE OF SCREEN-SPLP-BY-EXISTING-SPLP

Argument Name	Usage	Type	Description
<i>dbsplp-con</i>	input and output	container	<i>dbsplp</i> objects
n/a	returned value	container	<i>dbsplp</i> objects

If all of the *dbsplp* objects are removed in the screening process a message is printed. Otherwise, the *dbsplp* container is submitted to the database whereby the results are written to the *splp* table. This latter process is implemented by the local Scheme function `submit-aux`, as previously discussed.

### Perform Source-multiplicity Processing [8.3.9]

This subprocess produces estimates of source-multiplicity parameters, including cepstral analysis measures, for the detected signals. Only origins that fall within a specified distance range are processed. The station-to-event distance criteria are set by the parameters *evch-smult-min-delta* and *evch-smult-max-delta*. In addition, the origin must be shallower than a maximum depth criterion set by the symbol *evch-smult-max-depth*. A source-multiplicity estimation processing recipe from the *initsite* object is used to compute these measures. This subprocess is performed by the *DFX/libfs* function `make-smult`, which actually performs the source multiplicity analysis. This function is called by the Scheme language code:

```
(set! smult (make-smult wfdata-con dbarrival-con smult-rec
  orid))
```

The function requires four arguments, which are described in Table 143.

**TABLE 143: FUNCTIONAL INTERFACE OF MAKE-SMULT**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>dbarrival-con</i>	input	container	<i>dbarrival</i> objects
<i>smult-rec</i>	input	GObj	<i>smult estimation processing recipe</i> object
<i>orid</i>	input	integer	unique origin identification number
n/a	returned value	GObj	<i>smult</i> object

## ▼ Seismic Processing

This function, `make-smult`, returns an *smult* object. If this object proves to be null a warning message is printed. Otherwise, processing continues with the extraction of a *dbceppks* container from the *smult* object. Then the local Scheme function `screen-ceppks-by-existing-ceppks` is called to compare the newly generated *dbceppks* datasets with the *ceppks* records that already exist in the database. A duplicate *dbceppks* is determined by matching the *orid* and *sta* attributes. Any duplicates found in the new datasets are removed from the container. This subprocess follows the general form described in “Screening Database Containers” on page 43. This function is called by the Scheme language code:

```
(screen-ceppks-by-existing-ceppks dbceppks-con)
```

The function requires one argument, which is described in Table 144.

**TABLE 144: FUNCTIONAL INTERFACE OF SCREEN-CEPPKS-BY-EXISTING-CEPPKS**

Argument Name	Usage	Type	Description
<i>dbceppks-con</i>	input and output	container	<i>ceppks</i> objects
n/a	returned value	container	<i>ceppks</i> objects

If all of the *dbceppks* objects are removed in the screening process a message is printed. Otherwise, the *dbceppks* container is submitted to the database whereby the results are written to the **ceppks** table. This latter process is implemented by the local Scheme function `submit-aux`, as previously discussed.

Processing continues with the extraction of a *dbspvar* container from the *smult* object. The local Scheme function `screen-spvar-by-existing-spvar` is called to compare the newly generated *dbspvar* datasets with the *spvar* records that already exist in the database. A duplicate *dbspvar* is determined by matching the *arid* attribute. Any duplicates found in the new datasets are removed from the container. This subprocess follows the conventions described in “Screening Database Containers” on page 43. This function is called by the Scheme language code:

```
(screen-spvar-by-existing-spvar dbspvar-con)
```



The function requires one argument, which is described in Table 145.

**TABLE 145: FUNCTIONAL INTERFACE OF SCREEN-SPVAR-BY-EXISTING-SPVAR**

Argument Name	Usage	Type	Description
<i>dbspvar-con</i>	input and output	container	<i>spvar</i> objects
n/a	returned value	container	<i>spvar</i> objects

If all of the *dbspvar* objects are removed in the screening process a message is printed. Otherwise, the *dbspvar* container is submitted to the database whereby the results are written to the *spvar* table. This latter process is implemented by the local Scheme function *submit-aux*, as previously discussed.

### Perform Complexity Processing [8.3.10]

This subprocess produces estimates of complexity parameters for the detected signals. Only origins that fall within a specified distance range are processed. The station-to-event distance criteria are set by parameters *evch-complexity-min-delta* and *evch-complexity-max-delta*. In addition, the phase must be found in the list *evch-complexity-phase-list*. Furthermore, the phase must not match any of the PcP phases in the list *evch-PcP-list* if the station-event-distance lies in the distance interval defined by *evch-PcP-min-delta* and *evch-PcP-max-delta*. Finally, there must be one or more beam recipes whose *beam-name* values are found in the list *evch-complexity-beamrec*. A complexity estimation processing recipe from the *initsite* object is used to compute these measures. This subprocess is performed by the local Scheme function *compute-complexity*. This function is called by the Scheme language code:

```
(set! dbcomplexity-con (compute-complexity wfdata-con
  complexity-dbarsoc-con complexity-beamrec dborigin
  dbsite complexity-rec))
```

The function requires six arguments, which are described in Table 146.

## ▼ Seismic Processing

TABLE 146: FUNCTIONAL INTERFACE OF COMPUTE-COMPLEXITY

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	wfdata objects
<i>complexity-dbarsoc-con</i>	input	container	<i>dbarsoc</i> objects for the arrival associations appropriate to complexity analysis
<i>complexity-beamrec</i>	input	GObj	<i>beam recipe</i> object for the arrival associations appropriate to complexity analysis
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>complexity-rec</i>	input	GObj	<i>complexity estimation processing recipe</i> object
n/a	returned value	container	<i>dbcomplexity</i> objects

The local Scheme function `compute-complexity` is a wrapper to the *DFX/lib-complexity* function `cal-complexity`, which actually performs the complexity analysis.

This function, `compute-complexity`, returns a *dbcomplexity* container. If this container proves to be empty a warning message is printed. Otherwise, processing continues with the calling of the local Scheme function `screen-complexity-by-existing-complexity`. The newly generated *dbcomplexity* datasets are compared with complexity records that already exist in the database. A duplicate *dbcomplexity* is determined by matching the *orid*, *sta*, and *phase* attributes. Any duplicates found in the new datasets are removed from the container. This sub-process follows the general form described in "Screening Database Containers" on page 43. This function is called by the Scheme language code:

```
(screen-complexity-by-existing-complexity
  dbcomplexity-con)
```

The function requires one argument, which is described in Table 147.

**TABLE 147: FUNCTIONAL INTERFACE OF SCREEN-COMPLEXITY-BY-EXISTING-COMPLEXITY**

Argument Name	Usage	Type	Description
<i>dbcomplexity-con</i>	input and output	container	<i>dbcomplexity</i> objects
n/a	returned value	container	<i>dbcomplexity</i> objects

If all of the *dbcomplexity* objects are removed in the screening process a message is printed. Otherwise, the *dbcomplexity* container is submitted to the database whereby the results are written to the **complexity** table. This latter process is implemented by the local Scheme function `submit-aux`, as previously discussed.

Following the completion of all attribute computations, this function, `process-origin-for-init-site`, invokes the global Scheme function `commit-db` to commit all submissions to the database in the cases where the calculations were successful. In addition, the global Scheme function `gc-wfdata-con` is called to clear any stale memory pointers associated with waveform sets.



## Chapter 5: Hydroacoustic Processing

This chapter describes the detailed design of *DFX* hydroacoustic processing and includes the following applications:

- Automatic Hydroacoustic Detection [9]
- Interactive Hydroacoustic Recall [10]
- Automatic Hydroacoustic Event Characterization [11]

## Chapter 5: Hydroacoustic Processing

### AUTOMATIC HYDROACOUSTIC DETECTION [9]

*Automatic Hydroacoustic Detection (HydroDet)* is a DFX Scheme application that detects transient hydroacoustic signals and measures features in the proximity of the detection. The source code resides in the file `DFX-hydro-detection.scm`. *HydroDet* is typically called in the automatic pipeline processing, but it can also be run offline. *HydroDet* is usually called by specifying the station, start-time, end-time, and the `DFX-detection.par` file (the same one used in seismic processing) on the command line. *HydroDet* makes detections using a STA/LTA type detector. For each detection, it computes a series of features specific to hydroacoustic signal processing. The results of *HydroDet* are written to the **arrival**, **detection**, and **hydro\_features** tables.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Make Detections* [9.3]
- Detailed Description of *Measure Hydroacoustic Detection Features* [9.6]

## General Description

*HydroDet* was designed to automatically detect arrivals at hydroacoustic stations and measure properties relevant to hydroacoustics. The detections are based on a running STA/LTA type detector. *HydroDet* attempts to measure properties of each detection in a suite of frequency bands. However, properties are only measured in frequency bands where the snr is greater than a preset threshold (separate from the detection threshold). The measurements include several arrival-time measures, energy distributions, and cepstral parameters. One of the time measurements is used as the overall time of arrival (the probability-weighted time in the highest-snr frequency band).

## Architecture

*HydroDet* consists of seven sequential processes (see Figure 34). The first process initializes site information by creating an *initsite* object. Then waveforms are acquired for the given station and time interval. Detections are made using the same STA/LTA detector used in seismic processing but with different window lengths. The detection times are then refined using an auto-regressive technique. Any redundant detections are removed in the *Screen Detections* process. The hydroacoustic features are estimated for the remaining detections, and the results are written to the database.

## ▼ Hydroacoustic Processing

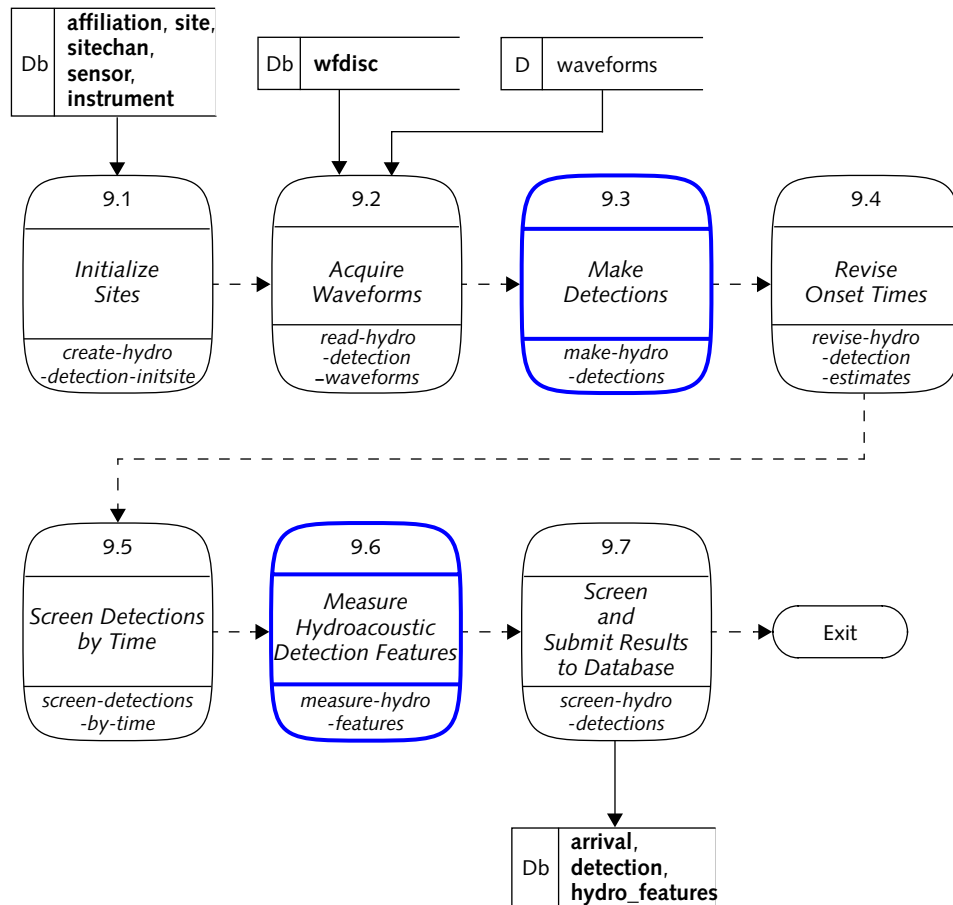


FIGURE 34. ARCHITECTURE OF HYDRODET

**Input/Output**

*HydroDet* is called for one station and time interval at a time. The station code, start-time, and end-time parameters are generally specified on the command line. The rest of the parameters are set or referenced in the `DFX-detection.par` file. Table 148 lists the input CVAR datastore parameters used in the Scheme code. This application loads the `DFX-site-detection.par` file, which in turn loads the necessary recipes (not listed here).



**TABLE 148: INPUT PARAMETERS FOR AUTOMATIC HYDROACOUSTIC DETECTION PROCESSING**

Par Name	Source	Default Value	Description
<i>sta</i>	command line	<none>	station code
<i>start-time</i>	command line	<none>	start-time of the processing interval
<i>end-time</i>	command line	<none>	end-time of the processing interval
<i>database-account</i>	par file	<none>	database access character-string
<i>out-arrival-table</i>	par file	arrival	output table containing arrival information generated by this application
<i>out-detection-table</i>	par file	detection	output table containing signal detection information generated by this application
<i>out-hydro_features-table</i>	par file	hydro_features	<b>hydro_features</b> table
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information
<i>site-table</i>	par file	site	table containing site-location information
<i>sitechan-table</i>	par file	sitechan	table containing station-channel information
<i>sensor-table</i>	par file	sensor	table containing specific calibration information for physical channels

## ▼ Hydroacoustic Processing

**TABLE 148: INPUT PARAMETERS FOR AUTOMATIC HYDROACOUSTIC DETECTION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>instrument-table</i>	par file	instrument	table containing generic (default) calibration information about a station
<i>affiliation-table</i>	par file	affiliation	table containing station and network affiliations
<i>wfdisc-extension-len</i>	par file	86400.0	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>NetType</i>	par file	<none>	network type, as in: "ss", "array", "larray"
<i>StaType</i>	par file	<none>	station type, as in: "1c", "3c"
<i>WaveType</i>	par file	" "	wave type (" ", "hydro-", "infra-")
<i>perform-qcstats</i>	par file	0 false	flag for generating and writing quality-control statistics: 0 means do not collect data quality statistics
<i>qcstats-scheme-file</i>	par file	<none>	filename containing the QCStats source code
<i>detection-site-recipe-file</i>	par file	<none>	filename of the site-specific detection recipes

Table 149 lists the output of *HydroDet*. *HydroDet* writes the results to the database, but first checks the database for detections that already exist by comparing arrival times. Those arrivals that do not already exist are written to the **detection**, **arrival** and **hydro\_features** tables. Table 238 on page 449 lists the specific attributes whose values are written by *HydroDet* for each of these database tables.

**TABLE 149: DATA PRODUCED BY AUTOMATIC HYDROACOUSTIC DETECTION PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>arrival</b>	database table	arrival attributes
<b>detection</b>	database table	detection attributes
<b>hydro_features</b>	database table	hydroacoustic detection features

## Processes

The following processes shown in Figure 34 on page 292 are described in this section:

- *Initialize Sites [9.1]*
- *Acquire Waveforms [9.2]*
- *Make Detections [9.3]*
- *Revise Onset Times [9.4]*
- *Screen Detections by Time [9.5]*
- *Measure Hydroacoustic Detection Features [9.6]*
- *Screen and Submit Results to Database [9.7]*

### Initialize Sites [9.1]

This process initializes site-specific information via the local Scheme function `create-hydro-detection-init-site`. This function is called by the Scheme code:

```
(set! init-site (create-hydro-detection-init-site sta))
```

## ▼ Hydroacoustic Processing

This local function follows the general site initialization model described in “Initializing Sites” on page 46. The function creates an *initsite* object with site recipes from the CVAR datastore and a *wfdata* container for the specified station and time interval.

The function has one input argument, which is described in Table 150. This argument is a character-string that contains the station code retrieved from the CVAR datastore prior to calling the function. The start-time and end-time for the interval are extracted from the CVAR datastore within the function along with the necessary table names and recipes.

The contents of the *initsite* object produced by the function are indicated in Table 8 on page 50.

**TABLE 150: FUNCTIONAL INTERFACE OF CREATE-HYDRO-DETECTION-INITSITE**

Argument Name	Usage	Type	Description
<i>sta</i>	input	character-string	station code
n/a	returned value	GObj	<i>initsite</i> object

### Acquire Waveforms [9.2]

This process acquires waveforms by the standard method used in *DFX* (see “Reading Waveforms” on page 58). It uses the local Scheme function `read-hydro-detection-waveforms`, which is called by the Scheme code:

```
(set! wfdata-con (read-hydro-detection-waveforms
  initsite))
```

The function has one argument, which is described in Table 151. The function takes an *initsite* object as input and produces a *wfdata* container. Two additional parameters that define the actual start-time and end-time of the data are added to the *initsite* object.

**TABLE 151: FUNCTIONAL INTERFACE OF READ-HYDRO-DETECTION-WAVEFORMS**

Argument Name	Usage	Type	Description
<i>initsite</i>	input and output	GObj	site-specific information
n/a	returned value	container	<i>wfdata</i> objects

The function performs the following steps:

1. Extract parameters from the CVAR datastore such as the user-requested start-time and end-time and the input **wfdisc** table name.
2. Extract recipes from the *initsite* object.
3. Determine the time interval.
4. Make a copy of the *wfdata* container from the *initsite* object.
5. Make a list of all station/channel pairs.
6. Create a *wfmem* container from the **wfdisc** table.
7. Read the waveforms.
8. Set the actual start-time and end-time in the *initsite* object.
9. Check the quality of the waveforms.
10. Return the *wfdata* container. Although the *wfmem* container is not explicitly returned by the function, its contents are accessible by the *wfdata* container.

### **Make Detections [9.3]**

This process makes detections via the local Scheme function `make-hydro-detections`. This function is called by the Scheme code:

```
(set! det-con (make-hydro-detections wfdata-con initsite))
```

## ▼ Hydroacoustic Processing

The function computes a running STA/LTA ratio over the filter bands specified in the beam recipes. A *trigger* object is created when the STA/LTA ratio exceeds a threshold (also specified in the beam recipe). These *trigger* objects are used to form *detection* objects (following rules specified by the detection recipe) that are then returned in a container.

The function has two arguments, which are described in Table 152. The function takes a *wfdata* container and an *initsite* object as input arguments and produces a *detection* container. The *detection* objects contain information indicating the trigger time, the best-snr beam, and an estimation of the time uncertainty.

**TABLE 152: FUNCTIONAL INTERFACE OF MAKE-HYDRO-DETECTIONS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>initsite</i>	input	GObj	site-specific information
n/a	returned value	container	<i>detection</i> objects

For more information on this function see “Detailed Description of Make Detections [9.3]” on page 304.

### Revise Onset Times [9.4]

This process uses an auto-regressive algorithm (see [IDC5.2.1]) to refine the onset times for each *detection* object in the container returned by the function *Make Detections*. The process uses the local Scheme function *revise-hydro-detection-estimates*, which is called by the Scheme code:

```
(revise-hydro-detection-estimates wfdata-con initsite
det-con)
```

The function has three arguments, which are described in Table 153. It takes the *wfdata* container, the *initsite* object, and the *detection* object as input arguments. The detection times are updated in the *detection* objects contained in *det-con*.

**TABLE 153: FUNCTIONAL INTERFACE OF REVISE-HYDRO-DETECTION-ESTIMATES**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>initsite</i>	input	GObj	site-specific information
<i>det-con</i>	input and output	container	<i>detection</i> objects
n/a	return value	container	<i>detection</i> objects

The function performs the following steps:

1. Extract the onset recipe from the *initsite* object.
2. Enter a loop over the objects in the *detection* container.
3. Call the *DFX/libonset* function *revise-onset* that performs the actual onset time revision.
4. Keep a list of detections that returned an error from *revise-onset*.
5. Exit the loop over the *detection* container.
6. Remove any detections that produced an error. This function returns a *detection* container.

### Screen Detections by Time [9.5]

This process screens redundant detections from the previous processes. It uses a global Scheme function, which is called by the Scheme code:

```
(screen-detections-by-time det-con
  det-min-detection-interval)
```

The function has two arguments, which are described in Table 154. It takes the *detection* container and the minimum allowed time interval as input. It then removes detections that are deemed as duplicates from the container. Detections are considered duplicated if they are within the minimum allowed time window of

## ▼ Hydroacoustic Processing

another detection. The time interval is specified in the CVAR datastore parameter *det-min-detection-interval*. On return, the input *detection* container is modified by the redundant detections being removed. The function also produces the screened *detection* container as the returned value. For more information on screening see "Internal Screening of Containers" on page 74.

**TABLE 154: FUNCTIONAL INTERFACE OF SCREEN-DETECTIONS-BY-TIME**

Argument Name	Usage	Type	Description
<i>det-con</i>	input and output	container	<i>detection</i> objects
<i>det-min-detection-interval</i>	input	real	minimum time allowed between detections
n/a	return value	container	screened <i>detection</i> objects

### Measure Hydroacoustic Detection Features [9.6]

This process estimates the hydroacoustic features for each detection over a set of filter bands. It uses the local Scheme function `measure-hydro-detection-features`, which is called by the Scheme code:

```
(set! dbhydro-con (measure-hydro-detection-features
  wfdata-con initsite det-con))
```

This function in turn calls the *DFX/libhydro* function `measure-hydro`. The features are measured for a set of filter bands specified by the filter-recipe table in the station-specific hydroacoustic par files. The detection attributes *time*, *deltim*, and *snr* are set using features from the filter band with the highest *snr*. The attributes *time* and *deltim* are set to the features *prob\_weight\_time* and *sigma\_time*, respectively.

The function `measure-hydro` has three arguments, which are described in Table 155. The function takes the *wfdata* container, the *initsite* object, and the *detection* container as input and produces a *dbhydro* container. The function also updates



the *time*, *deltim*, *snr* and *arid* values in all of the objects in the *detection* container. The *dbhydro* container has a *dbhydro* object for each unique detection/filter-band pair. A *dbhydro* object corresponds to the **hydro\_features** table.

For more information about this function see “Detailed Description of Measure Hydroacoustic Detection Features [9.6]” on page 307.

**TABLE 155: FUNCTIONAL INTERFACE OF MEASURE-HYDRO-DETECTION-FEATURES**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>initsite</i>	input	GObj	site-specific information
<i>det-con</i>	input and output	container	<i>detection</i> objects
n/a	returned value	container	<i>dbhydro</i> objects

#### Screen and Submit Results to Database [9.7]

This process passes the detections through a final screening process to ensure that the new detections are not duplicating detections already in the database. The process uses a local Scheme function, which is called by the Scheme language code:

```
(screen-hydro-detections initsite det-con dbhydro-con)
```

This function has three arguments, which are described in Table 156. It follows the conventions described in “Screening Database Containers” on page 43. The function takes the *initsite* object, the *detection* container, and the *dbhydro* container as input. Any detections that are found to already exist in the database are removed from the *detection* and *dbhydro* containers.

## ▼ Hydroacoustic Processing

TABLE 156: FUNCTIONAL INTERFACE OF SCREEN-HYDRO-DETECTIONS

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	site-specific information
<i>det-con</i>	input and output	container	<i>detection</i> objects
<i>dbhydro-con</i>	input and output	container	<i>dbhydro</i> objects
n/a	returned value	container	<i>dbhydro</i> objects

The function performs the following steps:

1. Extract the parameters, such as *det-min-detection-interval* in the *detection* recipe, from the *initsite* object.
2. Screen detections that are too close in time (see “Screen Detections by Time [9.5]” on page 299).
3. Screen detections by existing detections.
4. Remove *dbhydro* objects whose corresponding detection was deleted in the previous step from the *dbhydro* container. This function has to internally screen the detections again because the arrival times were revised when the hydroacoustic features were estimated.

The remaining detections are written to the database by a local *DFX* Scheme function. This function is called by the Scheme language code:

```
(submit-db-hydro-results det-con dbhydro-con)
```

The function has two arguments, which are described in Table 157. It takes the *detection* and *dbhydro* containers as input. It writes the appropriate attributes for each detection to the **arrival** and **detection** tables, and it writes the corresponding features to the **hydro\_features** table in the database.

TABLE 157: FUNCTIONAL INTERFACE OF SUBMIT-DB-HYDRO-RESULTS

Argument Name	Usage	Type	Description
<i>det-con</i>	input	container	<i>detection</i> objects
<i>dbhydro-con</i>	input	container	<i>dbhydro</i> objects
n/a	returned value	container	<i>detection</i> objects

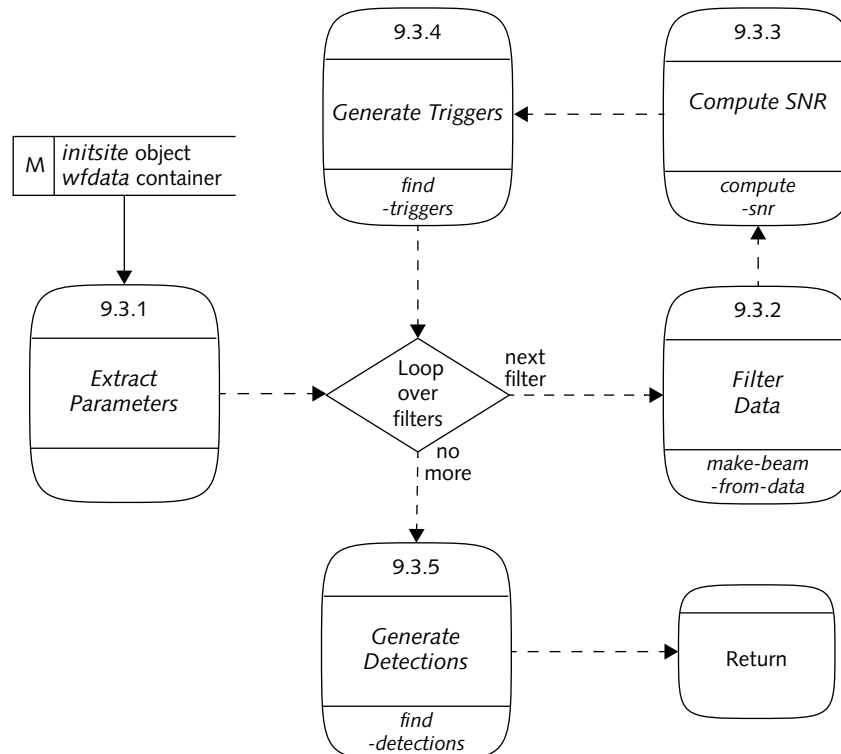
The function performs the following steps:

1. Extract the **arrival**, **detection**, and **hydro\_features** table names from the CVAR datastore.
2. Enter the loop over detections.
3. Create a *dbarrival* object for the current detection via `create-dbar-rival-from-detection`.
4. Submit the *dbarrival* object to the database via `submit-object-db`.
5. Create a *dbdetection* object for the current detection via `create-dbde-tection-from-detection`.
6. Submit the *dbdetection* object to the database.
7. Create a *dbhydro* container for a single detection from the input *dbhydro* container.
8. Submit the newly created *dbhydro* container to the database via `submit-container-db`.
9. If no errors occurred, save database additions via `commit-db`; otherwise, remove the additions via `rollback-db`.
10. Leave the loop over detections, and return the input detections con-tainer.

## ▼ Hydroacoustic Processing

**Detailed Description of Make  
Detections [9.3]**

Detections are determined using an STA/LTA type detector over a variety of frequency bands. The architecture of the function `make-hydro-detections` is shown in Figure 35. The function extracts various parameters including the beam recipe. The beam recipe specifies a set of filter bands. The function loops over each filter band and applies it to the waveforms. The STA and LTA time series are computed. Triggers are created when the STA/LTA ratio exceeds a threshold, specified independently for each filter in the beam recipe. After the filters have been exhausted, the triggers are used to find detections. Often there are a set of triggers over different frequency bands but nearby in time. These are coalesced into a single detection. The detections are returned in a *detection* container.

**FIGURE 35. ARCHITECTURE OF MAKE DETECTIONS**

The following subprocesses shown in Figure 35 are described in this section:

- *Extract Parameters [9.3.1]*
- *Filter Data [9.3.2]*
- *Compute SNR [9.3.3]*
- *Generate Triggers [9.3.4]*
- *Generate Detections [9.3.5]*

### **Extract Parameters [9.3.1]**

This subprocess extracts recipes from the *initsite* container via inline Scheme code. The function requires the beam and detection recipes as well as the start-time and end-time of the interval. The beam recipe set is only a list of filters and thresholds, because hydroacoustic processing is currently performed on single-sensor stations, and beam steering information is not applicable.

### **Filter Data [9.3.2]**

This subprocess filters the waveforms according to the current beam recipe via the global Scheme function `make-beam-from-data`. This function is called by the Scheme code:

```
(set! beam (make-beam-from-data wfdata-con beamrec))
```

The function has two arguments, which are described in Table 158. It takes the *wfdata* container and a *beam recipe* object as input and returns a *beam* object containing the filtered waveforms. This function is a wrapper to the *DFX/libbeam* function `make-beam`. For more information on filtering waveforms see [IDC5.2.1].

## ▼ Hydroacoustic Processing

TABLE 158: FUNCTIONAL INTERFACE OF MAKE-BEAM-FROM-DATA

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>beamrec</i>	input	GObj	<i>beam recipe</i> object that describes filter parameters
n/a	returned value	GObj	<i>beam</i> object containing screened time-series

**Compute SNR [9.3.3]**

This subprocess creates STA, LTA, and snr time series for the screened waveform. It uses the *DFX/libdetect* function `compute-snr`, which is called by the Scheme language code:

```
(compute-snr beam det-rec)
```

The function takes the screened data, which are stored in the *beam* object, and the *detection recipe* object as input. The STA, LTA, and snr time series are returned in the *beam* object. See “Generate STA and LTA [1.3.4]” on page 105 for more information about this function. See [IDC5.2.1] for more information about the algorithms used to compute STA and LTA time series.

**Generate Triggers [9.3.4]**

This subprocess finds the times that the STA/LTA ratio rises above the threshold specified in the beam recipe. It uses the *DFX/libdetect* function `find-triggers`, which is called by the Scheme language code:

```
(find-triggers beam start-time end-time trig-con det-rec)
```

The function has five arguments, which are described in Table 159. It takes the *beam* object, the start-time, the end-time, and the detection recipe as input. The function finds the times corresponding to points in the screened waveforms that exceed the thresholds defined in the detection recipe. The times are stored in the returned trigger container.

TABLE 159: FUNCTIONAL INTERFACE OF FIND-TRIGGERS

Argument Name	Usage	Type	Description
<i>beam</i>	input	GObj	<i>beam</i> object containing STA and LTA time-series
<i>start-time</i>	input	real	start-time of the processing interval
<i>end-time</i>	input	real	end-time of the processing interval
<i>trig-con</i>	output	container	<i>trigger</i> objects
<i>det-rec</i>	input	GObj	<i>detection recipe</i> object containing trigger thresholds
n/a	returned value	container	<i>trigger</i> objects

### Generate Detections [9.3.5]

After the process has looped through the filter list, the triggers are passed to this subprocess, which creates the actual detections. The subprocess uses the *DFX/lib-detect* function *find-detections*, which is called by the Scheme language code:

```
(find-detections trig-con det-rec)
```

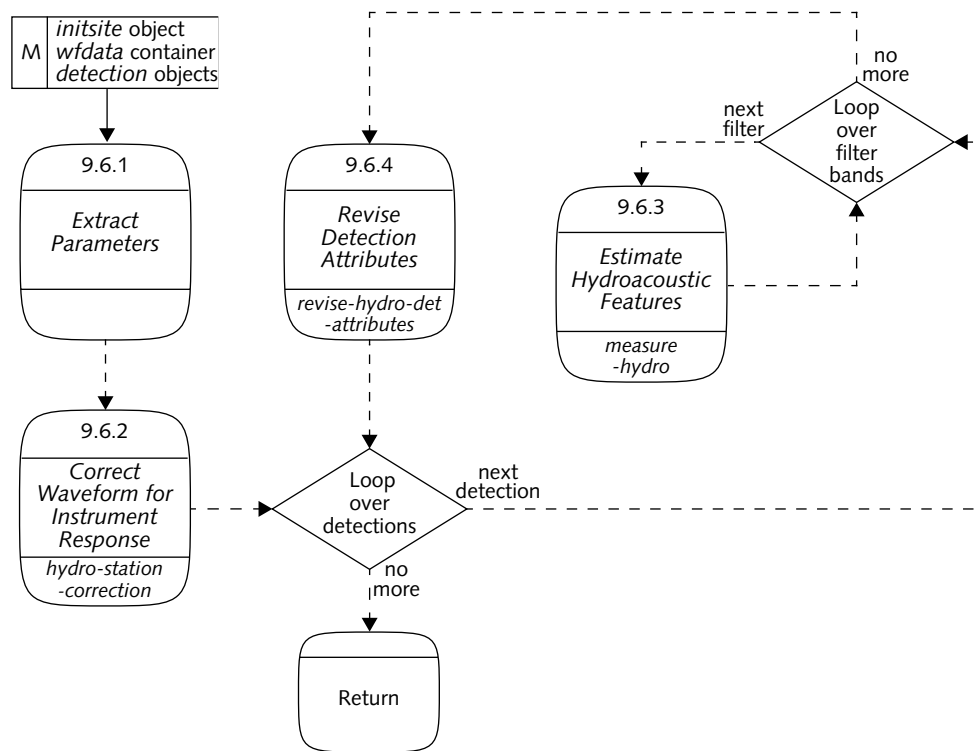
The triggers are processed and evaluated to form detections. This subprocess also estimates the time uncertainty stored in the attribute *deltim*, but this value is updated later in onset time revision. The detections are returned in a *detection* container. For more information see “Evaluate Triggers and Generate Detections [1.5.6]” on page 119.

### Detailed Description of Measure Hydroacoustic Detection Features [9.6]

This process makes the actual measurements that are placed into the **hydro\_features** database table. The process is accomplished by the function *measure-hydro-detection-features*. Figure 36 shows the architecture of this function. The

## ▼ Hydroacoustic Processing

appropriate parameters and filter bands are extracted from the input containers. The waveforms are corrected for the instrument response. The actual feature estimation occurs within a nested pair of loops. The outer loop is over detections, and the inner loop is over frequency bands. The hydroacoustic features are estimated and inserted into a *dbhydro* object. The final arrival time is found after looping over the filter bands.



**FIGURE 36. ARCHITECTURE OF MEASURE HYDROACOUSTIC DETECTION FEATURES**

The following subprocesses shown in Figure 36 are described in this section:

- *Extract Parameters [9.6.1]*
- *Correct Waveform for Instrument Response [9.6.2]*
- *Estimate Hydroacoustic Features [9.6.3]*



■ *Revise Detection Attributes [9.6.4]*

### **Extract Parameters [9.6.1]**

This subprocess extracts and initializes needed parameters from the *initsite* object and the CVAR datastore. The hydroacoustic recipe is retrieved from the *initsite* object, and the *dbhydro* container is initialized by inline Scheme code. The hydroacoustic recipe contains parameters used in the *DFX/libhydro* function *measure-hydro*. The filter list for the hydroacoustic features computations is extracted from the filter-recipe parameter in the CVAR datastore by the local *DFX* Scheme function *get-filter-list-from-cvar*. This filter list is separate from the beam-recipe filters used to find the detections.

### **Correct Waveform for Instrument Response [9.6.2]**

This subprocess removes the instrument response from the waveforms by using an inverse response filter if one exists for the station. The filter is a FIR filter that has been designed to remove instrument effects over the pass band of interest (typically 2–80 Hz). The filter must already exist in a par file in the *precond* subdirectory of the *DFX* configuration directory tree. The FIR filter coefficients are passed to the function through the CVAR datastore. The filter is applied to the data by the *DFX/libhydro* function *hydro-station-correction*. If the filter exists, the raw waveforms in the *wfdata* container are replaced with the station-corrected data. If the filter does not exist, the raw waveforms are left unchanged.

### **Estimate Hydroacoustic Features [9.6.3]**

This subprocess estimates the hydroacoustic features for each detection. It uses the *DFX/libhydro* function *measure-hydro*, which is called by the Scheme code:

```
(set! hydro (measure-hydro wfdata-con hydro-rec time))
```

The function has three arguments, which are described in Table 160. The only parameter that the function needs from the current *detection* object is the time. The hydroacoustic recipe contains two parameters, *hydro-flo* and *hydro-fhi*, that

## ▼ Hydroacoustic Processing

define the bandpass filter. These parameters are set from the filter recipe in the CVAR datastore prior to calling `measure-hydro`. The function returns a generic object of type *hydro* that contains the various feature estimates. For more information about hydroacoustic features see [Lan97g].

**TABLE 160: FUNCTIONAL INTERFACE OF MEASURE-HYDRO**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>hydro-rec</i>	input	GObj	<i>hydro recipe</i> object
<i>time</i>	input	real	detection time
n/a	returned value	GObj	<i>hydro</i> object

The *hydro* object returned from `measure-hydro` creates a *dbhydro* object suitable for insertion into the **hydro\_features** table. This is accomplished by the global Scheme function `create-dbhydro-from-det`. The *arid* attribute in the *dbhydro* object is taken from the *arid* in the *detection* object. A given detection has a *dbhydro* object for each filter band. These objects are placed in a container that is returned by `measure-hydro-detection-features`.

**Revise Detection Attributes [9.6.4]**

This subprocess uses the estimated hydroacoustic features to determine the final time of the *detection* object as well as the detection's overall snr. It uses the local Scheme function `revise-hydro-det-attributes`, which is called by the Scheme code:

```
(revise-hydro-det-attributes hydro-rec det
  sub-dbhydro-con)
```

The function has three arguments, which are described in Table 161. The function loops over the *hydro* objects for each band to find the greatest snr value. The snr is defined as follows:

$$\text{snr} = \frac{\text{total energy}}{\text{duration} \cdot \text{average noise}}$$

where both total energy and average noise have been converted from decibels to physical units in power. The *snr* attribute in the *detection* object is set to this maximum snr. The *time* attribute in the *detection* object is set to the *prob\_weight\_time* (probability-weighted time, one of the hydroacoustic features) of the frequency band with the maximum snr. The *deltim* attribute in the *detection* object is set to the *sigma\_time* (time uncertainty estimate, one of the hydroacoustic features) in the same frequency band. After setting these values the function returns a *detection* object with revised parameter values.

**TABLE 161: FUNCTIONAL INTERFACE OF REVISE-HYDRO-DET-ATTRIBUTES**

Argument Name	Usage	Type	Description
<i>hydro-rec</i>	input	GObj	<i>hydro recipe</i> object
<i>det</i>	input and output	GObj	<i>detection</i> object
<i>sub-dbhydro-con</i>	input	container	<i>dbhydro</i> objects
n/a	returned value	GObj	<i>detection</i> object

## ▼ Hydroacoustic Processing

**INTERACTIVE HYDROACOUSTIC  
RECALL [10]**

*Interactive Hydroacoustic Recall (IntHydroRcl)* is a DFX Scheme application that makes hydroacoustic feature measurements on analyst-added or modified arrivals. These are the same features computed by the *Automatic Hydroacoustic Detection* application. The application source code for *IntHydroRcl* resides in the file `DFX-hydro-recall.scm`. *IntHydroRcl* is typically called from within *ARS*, which sends an IPC message containing the names of temporary database tables that hold the arrivals to be reprocessed. *IntHydroRcl* computes the hydroacoustic features for each arrival using the onset and termination times and the filter chosen by the analyst. *IntHydroRcl* writes the new measurements to the temporary `hydro_features` table before exiting.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Perform Recall Signal Processing* [10.4]

**General Description**

*IntHydroRcl* was designed so that new hydroacoustic features could be measured after the analyst modifies the onset and termination times of a hydroacoustic signal (or adds a new arrival). The measurements include arrival times, energy distributions, and cepstral parameters. The estimation process is similar to that in *Automatic Hydroacoustic Detection* [9] (see page 290) except it does not attempt to calculate an onset and termination time because these are supplied by the analyst

through *ARS*. Another difference is that *IntHydroRcl* computes features for only one analyst-chosen frequency band. The *time*, *deltim*, and *auth* attributes are updated in the **arrival** table.

## Architecture

*IntHydroRcl* consists of four processes (see Figure 37). The *IntHydroRcl* Scheme code was designed as a pair of nested loops. The outer loop runs over stations, and the inner loop runs over arrivals. *IntHydroRcl* first retrieves the current arrivals and hydroacoustic features from the temporary tables created in *ARS*. It then forms a list of all stations covered by the arrivals. This list is used for the outer loop. An *initsite* object is created for each station and is used later to read waveforms and to provide parameters for feature measurements. The process inside the inner loop estimates the hydroacoustic features and writes the results to the database.

## Input/Output

*IntHydroRcl* receives input from and sends output to *ARS* through the contents of temporary database tables. The **arrival**, **detection**, and **hydro\_features** table names are given as command line parameters. The rest of the parameters are set or referenced in the `DFX-hydro-recall.par` file in the *DFX* par directory. The CVAR datastore parameters used by *IntHydroRcl* are listed in Table 162. The application loads the `DFX-site-detection.par` file, which in turn loads the necessary recipes (not listed here).

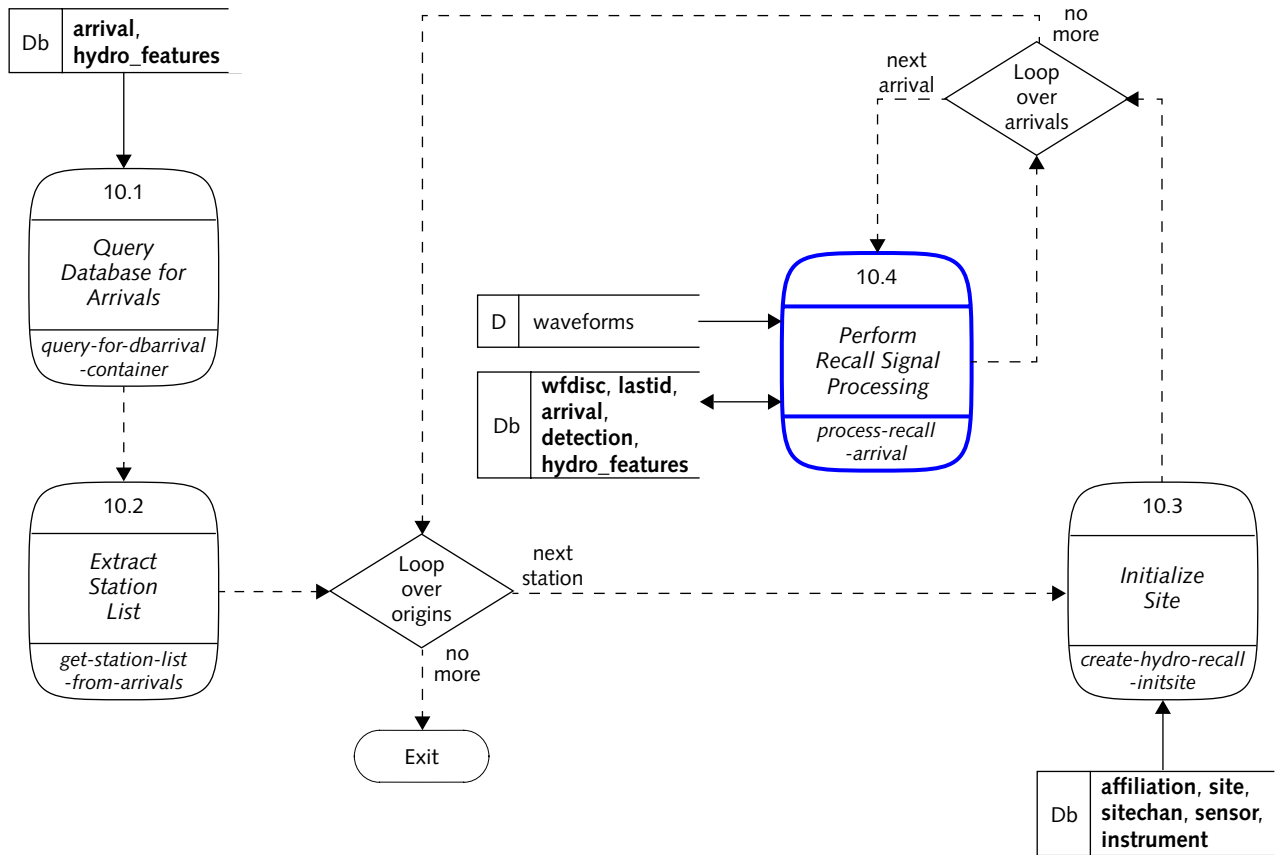


FIGURE 37. ARCHITECTURE OF INTHYDORCL

**TABLE 162: INPUT PARAMETERS FOR INTERACTIVE HYDROACOUSTIC RECALL PROCESSING**

Par Name	Source	Default Value	Description
<i>database-account</i>	command line	<none>	database access character-string
<i>arrival-table</i>	command line	<none>	<b>arrival</b> table
<i>detection-table</i>	command line	<none>	<b>detection</b> table
<i>hydro_features-table</i>	command line	<none>	<b>hydro_features</b> table
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information
<i>site-table</i>	par file	site	table containing site-location information
<i>sitechan-table</i>	par file	sitechan	table containing station-channel information
<i>sensor-table</i>	par file	sensor	table containing specific calibration information for physical channels
<i>instrument-table</i>	par file	instrument	table containing generic (default) calibration information about a station
<i>affiliation-table</i>	par file	affiliation	table containing station and network affiliations
<i>wfdisc-extension-len</i>	par file	86400.0	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>NetType</i>	par file	<none>	network type, as in: "ss", "array", "larray"

## ▼ Hydroacoustic Processing

**TABLE 162: INPUT PARAMETERS FOR INTERACTIVE HYDROACOUSTIC RECALL PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>StaType</i>	par file	<none>	station type, as in: "1c", "3c"
<i>WaveType</i>	par file	" "	wave type (" ", "hydro-", "infra-")
<i>recall-auth</i>	par file	<none>	character-string identifying process

The results of *IntHydroRcl* are listed in Table 163. *IntHydroRcl* updates all the attributes in **hydro\_features** except the onset and termination times and the filter parameters. It also updates the *time* and *deltim* attributes in the **arrival** and **detection** tables. Table 239 on page 450 lists the specific attributes whose values are written by *IntHydroRcl* for each of these database tables.

**TABLE 163: DATA PRODUCED BY INTERACTIVE HYDROACOUSTIC RECALL PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>arrival</b>	database table	updates <i>time</i> , <i>deltim</i> , and <i>auth</i> attributes
<b>detection</b>	database table	updates <i>time</i> and <i>deltim</i> attributes
<b>hydro_features</b>	database table	updates all attributes except <i>onset_time</i> , <i>termination_time</i> , and the filter parameters



## Processes

The following processes shown in Figure 37 on page 314 are described in this section:

- *Query Database for Arrivals [10.1]*
- *Extract Station List [10.2]*
- *Initialize Site [10.3]*
- *Perform Recall Signal Processing [10.4]*

### Query Database for Arrivals [10.1]

This process extracts the arrival information from the database and places it into a `dbarrival` container. It uses the local *DFX* Scheme function `query-for-dbar-rival-container`, which has no input parameters. Instead it obtains the **arrival** table name directly from the CVAR datastore. The **arrival** table is assumed to contain only the arrivals of interest; therefore, *IntHydroRcl* retrieves the entire **arrival** table. For information about how database queries are implemented see “Querying the Database” on page 41.

### Extract Station List [10.2]

This process creates a list of unique station names in the **arrival** table via the local Scheme function `get-station-list-from-arrivals`. The outer loop of *IntHydroRcl* (see Figure 37 on page 314) runs over each station in this list.

### Initialize Site [10.3]

This process initializes the site via the local Scheme function `create-hydro-recall-initsite`. The function is called by the Scheme code:

```
(set! initsite (create-hydro-recall-initsite sta
sta-dbararrival-con))
```

## ▼ Hydroacoustic Processing

This is a special case of the site initialization function (see “Initializing Sites” on page 46). The function creates an *initsite* object with site-specific recipes that are used later to get waveforms and estimate hydroacoustic features. The *dbarrival* container that is passed into the function is used to determine the data time interval used to create the *wfdata* container.

The function `create-hydro-recall-initsite` has two arguments, which are described in Table 164. It takes *sta* and *sta-dbarrival-con* as input parameters. The function is called for a single station specified by the station code *sta* (the current station of the outer loop). The arrivals for this station have been extracted and placed in their own container, *sta-dbarrival-con*.

**TABLE 164: FUNCTIONAL INTERFACE OF CREATE-HYDRO-RECALL-INITSITE**

Argument Name	Usage	Type	Description
<i>sta</i>	input	character-string	station code
<i>sta-dbarrival-con</i>	input	container	<i>dbarrival</i> objects for this station
n/a	returned value	GObj	site-specific information

The Scheme function returns an *initsite* object. The contents of the *initsite* object are indicated in Table 8 on page 50. The *initsite* container holds all of the information needed to acquire and process the data for this particular station.

### Perform Recall Signal Processing [10.4]

This process performs the actual feature estimation for each arrival. It uses the local Scheme function `process-recall-arrival`, which is called by the Scheme code:

```
(process-recall-arrival initsite dbarrival dbhydro)
```

The function is executed for each arrival belonging to the current station. The function's tasks include reading the waveforms, correcting the data for instrument response, and estimating the hydroacoustic features. It also writes the results to the appropriate tables in the database.

The function has three arguments, which are described in Table 165. There are three input objects: *initsite*, *dbarrival*, and *dbhydro*. The *dbarrival* object contains the current arrival information. The *dbhydro* object contains the analyst-specified onset and termination times and the filter parameters.

**TABLE 165: FUNCTIONAL INTERFACE OF PROCESS-RECALL-ARRIVAL**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	site-specific information
<i>dbarrival</i>	input	GObj	<i>dbarrival</i> object
<i>dbhydro</i>	input	GObj	<i>dbhydro</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The function `process-recall-arrival` writes the new hydroacoustic features to the database and updates the arrival and detection times. This function is described in “Detailed Description of Perform Recall Signal Processing [10.4].”

### Detailed Description of Perform Recall Signal Processing [10.4]

The recall processing in the function `process-recall-arrival` occurs as four subprocesses (see Figure 38). First, a *detection* object is created from the *dbarrival* object. Then this object is used to read the waveforms. The instrument response, if available, is removed from the waveforms. The final subprocesses estimate the hydroacoustic features and write the results to the database.

## ▼ Hydroacoustic Processing

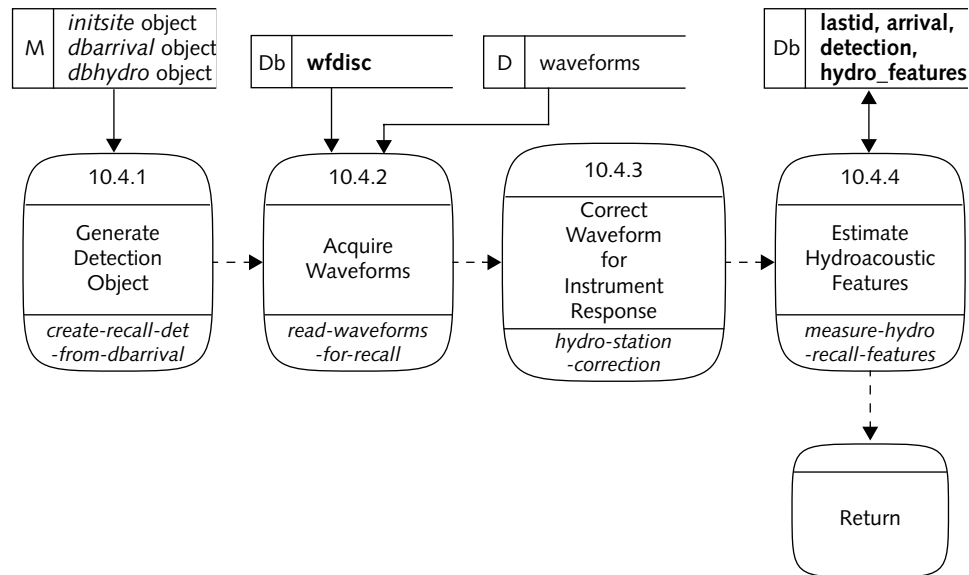


FIGURE 38. ARCHITECTURE OF PERFORM RECALL SIGNAL PROCESSING

The following subprocesses shown in Figure 38 are described in this section:

- *Generate Detection Object [10.4.1]*
- *Acquire Waveforms [10.4.2]*
- *Correct Waveform for Instrument Response [10.4.3]*
- *Estimate Hydroacoustic Features [10.4.4]*

**Generate Detection Object [10.4.1]**

This subprocess creates a detection object from the dbarrival object via the global Scheme function `create-recall-det-from-dbarrival`. The relevant information is extracted from the dbarrival object, converted to new units if necessary, and placed into a detection object, which is returned by the function.

### Acquire Waveforms [10.4.2]

This subprocess acquires waveforms via a local Scheme function `read-waveforms-for-recall`. See “Reading Waveforms” on page 58 for a general description of waveform acquisition. The function determines an appropriate time interval by using the detection recipe parameters and the arrival time to compute data start-time and end-times. This creates a minimum time interval that still provides mitigation of edge effects from filtering and other processing. The waveforms are returned in a *wfdata* container.

### Correct Waveform for Instrument Response [10.4.3]

This subprocess removes the instrument response, if available, from the waveforms by using an inverse response filter if one exists for the station. The filter is a FIR filter that has been designed to remove instrument effects over the pass band of interest (typically 2–80 Hz). The filter must already exist in a *par* file in the *pre-cond* subdirectory. The FIR filter is stored in the CVAR datastore. The filter is applied to the data by the *DFX/libhydro* function `hydro-station-correction`. If the filter exists, the raw waveforms in the *wfdata* container are replaced with the corrected data. If the filter does not exist, the raw waveforms are left unchanged.

### Estimate Hydroacoustic Features [10.4.4]

This subprocess estimates the new features for the current arrival. It uses the local Scheme function `measure-hydro-recall-features`, which is called by the Scheme code:

```
(measure-hydro-recall-features wfdata-con initsite det
dbhydro)
```

This function has four arguments, which are described in Table 166. The function takes the waveforms, site information, *detection* object, and *hydro* object as input. The *detection* and *dbhydro* objects are the ones obtained from ARS via the temporary tables. On return, the function updates the features in the *dbhydro* object.

## ▼ Hydroacoustic Processing

**TABLE 166: FUNCTIONAL INTERFACE OF MEASURE-HYDRO-RECALL-  
FEATURES**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>initsite</i>	input	GObj	station-specific information
<i>det</i>	input	GObj	<i>detection</i> object
<i>dbhydro</i>	input and output	GObj	<i>dbhydro</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The function `measure-hydro-recall-function` creates a *hydro* object using the *DFX/libhydro* function `remeasure-hydro`. This function is similar to `measure-hydro` (described in “Automatic Hydroacoustic Detection [9]” on page 290) except the onset and termination times are provided from the *dbhydro* object. The returned *hydro* object is the same as that returned by `measure-hydro`.

The *hydro* and *detection* objects are used to create new *dbhydro*, *dbarrival*, and *dbdetection* objects. These are then used to update the records in the input database tables.

## AUTOMATIC HYDROACOUSTIC EVENT CHARACTERIZATION [11]

*Automatic Hydroacoustic Event Characterization (HydroEvch)* is a DFX Scheme application that makes amplitude measurements of hydroacoustic waveforms at predicted signal arrival times. The source code resides in the file `DFX-evch-hydro.scm`. *HydroEvch* is typically called in the automatic post-analyst processing pipeline. *HydroEvch* is executed by specifying the database, start-time, end-time, network, and the `DFX-evch-hydro.par` file. The predicted arrival and measured amplitudes are written to the **pararrival** and **amplitude** tables in the database. The results of this processing are used in *Event Screening*.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Estimate Event Characteristics* [11.3]

### General Description

*HydroEvch* was designed to estimate the maximum possible hydroacoustic signal at each hydrophone station for each event in the REB that could reasonably produce a hydroacoustic signal [WGB00c]. It also estimates the noise level for the predicted time of arrival, which provides a means of validating the signal integrity. For each origin in the time interval, *HydroEvch* finds all hydrophone stations in the network that have a clear water path between the station and event. It predicts an arrival time window at each of these stations based on the origin error ellipse and hydroacoustic travel-time tables. *HydroEvch* computes up to four amplitude measures

## ▼ Hydroacoustic Processing

within the predicted time window. These amplitude measures are designed to estimate the noise level and the maximum signal level in the time window in two frequency bands. The results are written to the **parrival** and **amplitude** tables.

**Architecture**

*HydroEvch* consists of three processes (see Figure 39). The main processing occurs within a nested pair of loops. The outer loop cycles over origins, and the inner loop cycles over the hydrophone stations. Prior to entering the loops, the function extracts the necessary input parameters from the CVAR datastore. *HydroEvch* then queries for the origins within the user-supplied start-time and end-time. The site information is also initialized prior to entering the nested pair of loops. The signal amplitudes are measured by the process within the inner loop. This process also writes the results to the database.



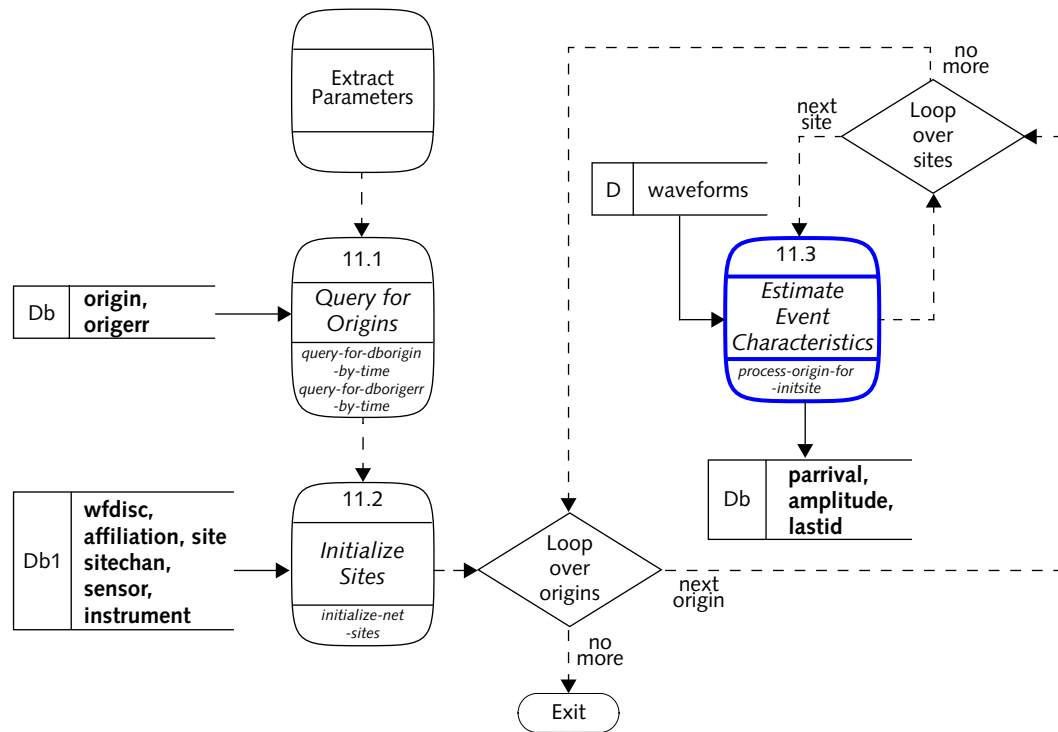


FIGURE 39. ARCHITECTURE OF HYDROEVCH

### Input/Output

*HydroEvch* is called for an interval of time and a given station network. These parameters are generally specified on the command line. The rest of the parameters are set or referenced in the `DFX-evch-hydro.par` file. The CVAR parameters used in the Scheme code are listed in Table 167.

## ▼ Hydroacoustic Processing

**TABLE 167: INPUT PARAMETERS FOR HYDROACOUSTIC EVENT CHARACTERIZATION PROCESSING**

Par Name	Source	Default Value	Description
<i>net</i>	command line	<none>	network of stations to process
<i>start-time</i>	command line	<none>	start-time of the processing interval
<i>end-time</i>	command line	<none>	end-time of the processing interval
<i>database-account</i>	par file	<none>	database access character-string
<i>origin-table</i>	par file	origin	input <b>origin</b> table
<i>origerr-table</i>	par file	origerr	input origin error table
<i>out-parrival-table</i>	par file	arrival	output table containing predicted arrival information generated by this application
<i>out-amplitude-table</i>	par file	amplitude	output table containing amplitude information generated by this application
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information
<i>site-table</i>	par file	site	table containing site-location information
<i>sitechann-table</i>	par file	sitechann	table containing station-channel information
<i>sensor-table</i>	par file	sensor	table containing specific calibration information for physical channels
<i>instrument-table</i>	par file	instrument	table containing generic (default) calibration information about a station

**TABLE 167: INPUT PARAMETERS FOR HYDROACOUSTIC EVENT CHARACTERIZATION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>affiliation-table</i>	par file	<i>affiliation</i>	table containing station and network affiliations
<i>wfdisc-extension-len</i>	par file	86400.0	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>evch-hydro-velocity</i>	par file	1.485	nominal sound speed in the ocean
<i>hydro-channels</i>	par file	<i>ep,sp</i>	channel codes for hydrophones
<i>hevch-phase-lead</i>	par file	60	extra time to ensure window includes any possible arrival
<i>hevch-phase-lag</i>	par file	60	extra time to ensure window includes any possible arrival
<i>hevch-data-offset-len</i>	par file	<none>	extra time to handle edge effects
<i>evch-lta-len</i>	par file	10	length of long-term-average window
<i>defeat-blockage-check</i>	par file	0	flag to turn blockage checking off
<i>blockage-spec-dir</i>	par file	<none>	directory containing blockage files
<i>hevch-site-recipe-file</i>	par file	<none>	par file pointing to site-specific recipe files

## ▼ Hydroacoustic Processing

Table 168 lists the output of *HydroEvch*. *HydroEvch* writes the predicted arrivals and measured amplitudes to the database, but it first checks the database for predicted arrivals that already exist in **parrival** by comparing origin and phase identifiers. Those predicted arrivals that do not already exist are written to the **parrival** and **amplitude** tables. Table 240 on page 450 lists the specific attributes whose values are written by *HydroEvch* for each of these database tables.

**TABLE 168: DATA PRODUCED BY HYDROACOUSTIC EVENT CHARACTERIZATION PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>parrival</b>	database table	predicted arrival attributes
<b>amplitude</b>	database table	amplitude and snr estimates

## Processes

The following processes shown in Figure 39 on page 325 are described in this section:

- *Query for Origins [11.1]*
- *Initialize Sites [11.2]*
- *Estimate Event Characteristics [11.3]*

### Query for Origins [11.1]

This process obtains the **origin** and **origerr** records from the database. The queries are constrained solely by the start-time and end-time set by the user. This process uses the local Scheme functions `query-for-dborigin-by-time` and `query-for-dborigerr-by-time`. The functions return *dborigin* and *dborigerr* containers. (See “Database Operations” on page 40 for more information about database queries.) The locations from *dborigin* are used to determine if an unblocked water

path exists between the origin and hydrophone and to predict the arrival time of hydroacoustic phases. The uncertainty information from *dborigerr* is used to determine the time window for measuring amplitudes.

### Initialize Sites [11.2]

This process initializes the site information via the local Scheme function `initialize-net-sites`. The function follows the general form described in “Initializing Sites” on page 46. It determines which stations are in the network and retrieves the station-specific information for each one. It also initializes the travel-time tables and determines the time interval for analysis. The corresponding *wfdata* and *wfmem* containers are read for each station (but the waveforms are not read at this stage). The function is called by the Scheme language code:

```
(set! initsite-con (initialize-net-sites dborigin-con
                                         dborigerr-con))
```

The function has two arguments, which are described in Table 169. The function returns an *initsite* container. The input containers are used only for determining the time interval. See Table 8 on page 50 for the contents of the returned *initsite* objects.

**TABLE 169: FUNCTIONAL INTERFACE OF INITIALIZE-NET-SITES**

Argument Name	Usage	Type	Description
<i>dborigin-con</i>	input	container	<i>dborigin</i> objects
<i>dborigerr-con</i>	input	container	<i>dborigerr</i> objects
n/a	returned value	container	<i>initsite</i> objects

### Estimate Event Characteristics [11.3]

Each origin/site pair is processed by the local Scheme function `process-origin-for-initsite`. This function checks for blockage, predicts arrival times, reads waveforms, estimates amplitude measures, and writes the results to the database. The function is called by the Scheme code:

## ▼ Hydroacoustic Processing

```
(process-origin-for-init-site init-site db-origin db-originerr)
```

The function `process-origin-for-init-site` has three arguments, which are described in Table 170. The *init-site* object for the current station has been extracted from the *init-site* container. The *db-origin* and *db-originerr* objects contain the information for the current origin.

**TABLE 170: FUNCTIONAL INTERFACE OF PROCESS-ORIGIN-FOR-INIT-SITE**

Argument Name	Usage	Type	Description
<i>init-site</i>	input	GObj	site-specific information
<i>db-origin</i>	input	GObj	<i>db-origin</i> object
<i>db-originerr</i>	input	GObj	<i>db-originerr</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The predicted arrivals and measured amplitudes are written to the **parrival** and **amplitude** tables specified in the CVAR datastore. The origin identifier (*orid*) links the origins to the predicted arrivals, and the amplitude identifier (*ampid*) links the predicted arrivals to the amplitude measurements. The returned value indicates success or failure within the function.

For more information on this function see the next section, Detailed Description of Estimate Event Characteristics [11.3].

### Detailed Description of Estimate Event Characteristics [11.3]

The hydroacoustic event characteristics are estimated for each origin and station. Figure 40 shows the architecture of the function `process-origin-for-init-site`. The function checks if there is an unblocked water path between the origin and station. If there is no unblocked path, the function immediately returns with no further computations. Processing continues for event/station pairs that have an unblocked path. The length of the time window to use for amplitude mea-

asures is determined by using the origin location uncertainty. The waveforms are read, and the predicted arrivals are formed. The amplitude measures occur inside a loop over the amplitude recipes. The results are screened for previously existing predicted arrivals. Those that pass are written to the database. These subprocesses are described in more detail in the following sections.

The following subprocesses shown in Figure 40 are described:

- *Check Station Blockage [11.3.1]*
- *Compute Time Interval [11.3.2]*
- *Acquire Waveforms [11.3.3]*
- *Predict Arrivals [11.3.4]*
- *Estimate Amplitudes [11.3.5]*
- *Screen and Submit Results [11.3.6]*

## ▼ Hydroacoustic Processing

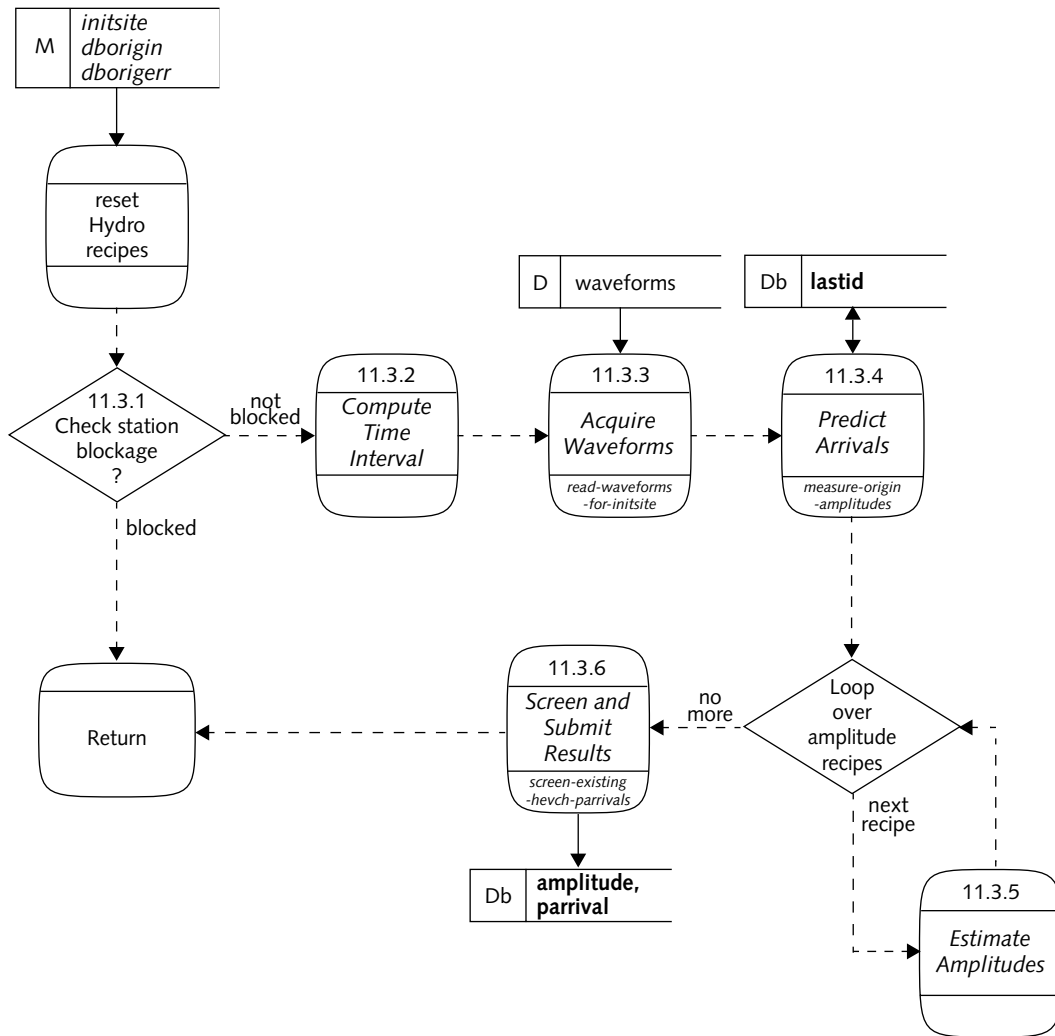


FIGURE 40. ARCHITECTURE OF ESTIMATE EVENT CHARACTERISTICS

**Check Station Blockage [11.3.1]**

The function `process-origin-for-initsite` measures amplitudes for event/station pairs that have an unblocked hydroacoustic path. The subprocess *Check Blockage* determines whether a given event has an unblocked hydroacoustic path



to a given station. This is accomplished with the local Scheme function `site-hydro-blocked`, which is a wrapper for the *DFX/libutil* function `ellipse-path-clear?`. The blockage check is liberal; if any part of the 90% error ellipse has an unblocked path then the station/event path is considered unblocked. This is less restrictive than the blockage check in Event Screening, which requires the entire ellipse to be unblocked. The user-parameter *skip-blockage-check* can be set to bypass the blockage check and hence compute predicted arrivals for all station/event pairs. The function is called by the Scheme language code:

```
(site-hydro-blocked dbsite dborigin dborigerr)
```

The function `site-hydro-blocked` has three arguments, which are described in Table 171. The function returns `nil` if there is an unblocked water path or if the parameter *defeat-blockage-check* has been set to a non-zero value in the CVAR. In all other cases the function returns `t` (for true).

**TABLE 171: FUNCTIONAL INTERFACE OF SITE-HYDRO-BLOCKED**

Argument Name	Usage	Type	Description
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>dborigerr</i>	input	GObj	<i>dborigerr</i> object
n/a	returned value	logical scalar	<code>t</code> (blocked) or <code>nil</code> (not blocked)

### Compute Time Interval [11.3.2]

This subprocess uses inline Scheme code to estimate a window of time that is large enough to include a hydroacoustic signal from the event, given the uncertainty in the location estimate. The time window length is determined by the length of the error ellipse's major axis and the nominal ocean's speed of sound (set in the CVAR datastore). This time is increased by an amount set in the CVAR parameters, *hevch-phase-lead* and *hevch-phase-lag*, to account for any systematic error that may not

## ▼ Hydroacoustic Processing

be represented in the error ellipse. Because the sound velocity in water is so slow as compared with seismic velocities, the error in origin time is not significant compared to the error in the location.

**Acquire Waveforms [11.3.3]**

This subprocess reads the waveforms for the time interval computed in *Compute Time Interval* and centered on the predicted arrival time of the hydroacoustic signal. It uses the local Scheme function `read-waveforms-for-initsite`, which reads the appropriate data, checks the quality of the data, and corrects for the instrument response of the station (see “Correct Waveform for Instrument Response [9.6.2]” on page 309). The function is called by the Scheme language code:

```
(set! wfdata-con (read-waveforms-for-initsite initsite
        dborigin dborigerr))
```

The function `read-waveforms-for-initsite` has three arguments, which are described in Table 172. The returned *wfdata* container points to the station-corrected waveforms.

**TABLE 172: FUNCTIONAL INTERFACE OF READ-WAVEFORMS-FOR-INITSITE**

Argument Name	Usage	Type	Description
<i>initsite</i>	input	GObj	site-specific information
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>dborigerr</i>	input	GObj	<i>dborigerr</i> object
n/a	returned value	container	<i>wfdata</i> objects

**Predict Arrivals [11.3.4]**

This subprocess creates the predicted arrival and amplitude database objects via the global DFX Scheme function `measure-origin-amplitudes`. The returned *dbparrival* and *dbamplitude* containers have the predicted arrival times, book-keeping information, and amplitude information. The *amp* attribute in the *dbamplitude*

container is filled with a measure defined in the amplitude recipe. The *stav* recipe computes the maximum STA, and the *mean-sqr* recipe computes the rms of the signal. The function is called by the Scheme language code:

```
(set! dbamplitude-dbarrival-list
  (measure-origin-amplitudes wfdata-con dbsite dborigin
    theo-amprec-con))
```

The function has four arguments, which are described in Table 173. The function returns a list that holds two containers. The first container has the predicted arrival objects, and the second container has the amplitude objects.

**TABLE 173: FUNCTIONAL INTERFACE OF MEASURE-ORIGIN-AMPLITUDES**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>theo-amprec-con</i>	input	container	<i>amplitude estimation recipe</i> objects
n/a	returned value	list	<i>dbarrival</i> and <i>dbamplitude</i> containers

### Estimate Amplitudes [11.3.5]

This subprocess makes the noise estimates, computes snrs, and converts amplitude measures into decibels. These tasks are executed by a set of Scheme functions that occur within a loop that runs over each *dbamplitude* object created in *Predict Arrivals*. The results are placed in the appropriate attributes of the *dbamplitude* object.

First, the noise estimates are made by two local Scheme functions, *compute-amp-beam-lta* and *compute-ampbeam-lta-squared*. The noise estimates are based on the minimum value of the LTA of the time series and the time series squared, respectively. These functions use the same two arguments, which are described in Table 174.

## ▼ Hydroacoustic Processing

**TABLE 174: FUNCTIONAL INTERFACE OF COMPUTE-AMPBEAM-LTA AND COMPUTE-AMPBEAM-LTA-SQUARED**

Argument Name	Usage	Type	Description
<i>amprec</i>	input	GObj	<i>amplitude estimation recipe</i> objects
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
n/a	returned value	real	minimum long-term average

Next, the appropriate STA for the amplitude type is retrieved from the *dbamplitude* container by the function `find-stav-measure-for-beam-name`. This function uses three arguments, which are described in Table 175.

**TABLE 175: FUNCTIONAL INTERFACE OF FIND-STAV-MEASURE-FOR-BEAM-NAME**

Argument Name	Usage	Type	Description
<i>beam-name</i>	input	character-string	beam name to match in <i>amplitude recipe</i> beam-name attribute
<i>dbamp-con</i>	input	container	<i>dbamplitude</i> objects
<i>amprec-con</i>	input	container	<i>amplitude estimation recipe</i> objects
n/a	returned value	real	short-term average

The *snr* is defined as the ratio of the maximum signal amplitude to the minimum LTA. It is set depending on the method used to compute the amplitude, as follows:

- If the amplitude type defined by the beam name is the maximum STA, then the minimum LTA level of the time series is used for the noise level.
- If the amplitude type is the mean-squared amplitude, then the minimum LTA level of the squared time series is used for the noise level.

Next, a set of inline Scheme code converts the amplitude and signal-to-noise measures into decibels. If the station has been calibrated, the units are dB relative to 1  $\mu$ Pa, otherwise the units are dB relative to 1 count.

Finally, the *amp* and *snr* attributes in the *dbamplitude* objects are reset to the newly calculated values. The *units* attribute is set to "dB rel 1 microPa" if the instrument response was corrected; otherwise, it is set to "dB rel 1 count".

### Screen and Submit Results [11.3.6]

This subprocess screens the newly computed **parrival** and **amplitude** records to ensure that they do not already exist in the database. A **parrival** record is considered a duplicate if the origin identifier, station code, and phase identifier for the predicted arrival matches the same attributes of a predicted arrival already in the database. The subprocess uses the local *DFX* Scheme function `screen-existing-hevch-parrivals`, which is called by the Scheme language code:

```
(screen-existing-hevch-parrivals dborigin dbparrival-con
dbamplitude-con parrival-table)
```

This function has four arguments, which are described in Table 176. On return, the *dbparrival* and *dbamplitude* containers only have objects that do not already exist in the database. The function itself returns nothing of interest. For more information see "Screening Database Containers" on page 43.

**TABLE 176: FUNCTIONAL INTERFACE OF SCREEN-EXISTING-HEVCH-PARRIVALS**

Argument Name	Usage	Type	Description
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>dbparrival-con</i>	input and output	container	<i>dbparrival</i> objects
<i>dbamplitude-con</i>	input and output	container	<i>dbamplitude</i> objects
<i>parrival-table</i>	input	character-string	<b>parrival</b> table

▼ **Hydroacoustic Processing**

The remaining objects in the parrival and amplitude database containers are submitted to the database. The subprocess uses a local Scheme function `submit-aux`, which is a wrapper for the global Scheme function `submit-container-db`. If an error is detected, the changes are discarded using the global Scheme function `rollback-db`. For more information on submitting results to the database see “Writing to the Database” on page 44.

## Chapter 6: Infrasonic Processing

This chapter describes the detailed design of *DFX* infrasonic processing and includes the following topic:

- Automatic Infrasonic Detection [12]

## Chapter 6: Infrasonic Processing

### AUTOMATIC INFRASONIC DETECTION [12]

*Automatic Infrasonic Detection (InfraDet)* is a DFX Scheme application that makes detections and measures features of signals recorded at the infrasonic sensor array stations. The application source code resides in the file `DFX-infra-detection.scm`. *InfraDet* is invoked as part of the automatic pipeline processing and is run on a given station for specified start-times and end-times. *InfraDet* makes detections using a two-stage process: first, in the Primary Detection Space (PDS), a spatial coherence detector is applied, then, in the Coincidence Detection Space (CDS), both an STA/LTA energy detector and a spatial coherence detector are applied. To declare an infrasonic detection the energy and spatial coherence detectors must coincide in exceeding their thresholds. For each detection, this application computes a series of features specific to infrasonic processing. The results of *InfraDet* are written to the **amplitude**, **arrival**, **detection**, **infra\_features**, and **wfdisc** tables. In addition, for each station and processing interval, three coherence traces are generated and saved as beams that can be displayed in ARS.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes



- Detailed Description of *Generate Primary Detection Space (PDS)* [12.3]
- Detailed Description of *Generate Coherence Traces as Beams* [12.4]
- Detailed Description of *Remove Redundant Detections* [12.6]
- Detailed Description of *Estimate Signal Amplitude* [12.7]
- Detailed Description of *Write Non-Redundant Detections* [12.8]

## General Description

This application performs three functions using infrasonic data: It declares detections, it estimates and saves waveform features, and it generates and saves coherence traces as beams that can be displayed in ARS. The detections are achieved in a two step process. First, a Primary Detection Space (PDS) is generated based upon spatial coherence and produces candidate detections. Second, a Coincidence Detection Space (CDS) is generated based upon the time overlap (coincidence) of both spatial coherence and energy excess. The CDS determines which detections are worthy of being declared and analyzed. For each declared detection, *InfraDet* estimates a variety of time-domain and frequency-domain waveform features that may be of use in characterizing the signal.

*InfraDet* also generates and saves (as beams) three coherence traces that can be displayed by ARS alongside the input waveforms. One of these traces provides a time-history of the maximum spatial coherence level (scaled to an equivalent F-statistic) achieved at each point in time. The other two traces provide the azimuth and the magnitude trace-velocity values (corresponding to the spatial coherence trace.) Thus, when displayed as a plot, the spatial coherence level, from time sample to time sample, gives a visual estimate of the time period where airborne infrasonic waves may have swept over the array. This is helpful to human analysts reviewing the results of automatic processing.

## Architecture

*InfraDet* is implemented via a single pass through a sequence of eight processes (see Figure 41). The process *Initialize Parameters* initializes a variety of processing recipes. This is followed by process *Acquire Waveforms*, which creates a container

▼ **Infrasonic Processing**

of *wfdata* objects that contain *wfmem* objects and quality-control masks. Then, the principal infrasonic detection processing is performed by the process *Generate Primary Detection Space (PDS)*. Following the generation and evaluation of the PDS, the next process *Generate Coherence Traces as Beams* creates and writes time series of measures related to the spatial coherence, which can be displayed as traditional beams in *ARS*. Records pointing to the coherence traces are written to the **wfdisc** table. The time-series samples of these traces are written as binary files to the UNIX filesystem. The next process *Generate Coincidence Detection Space (CDS)* declares detections where there is overlap in time (1) above threshold spatial coherence in a given vector-slowness and (2) above threshold occurrence of beam snr with corresponding vector-slowness. Where there is a declared detection, a variety of features are extracted from the waveforms in the vicinity of the detection. The entire set of declared detections are then evaluated to eliminate redundant detections. This is performed by the process *Remove Redundant Detections*. For each remaining detection, an amplitude estimate of the signal is extracted. This is performed by the process *Estimate Signal Amplitude*. Finally, the detections are written to the **amplitude**, **arrival**, **detection**, and **infra\_features** database tables by the process *Write Non-redundant Detections*.

**Input/Output**

Table 177 lists the input CVAR datastore parameters used in the Scheme code.

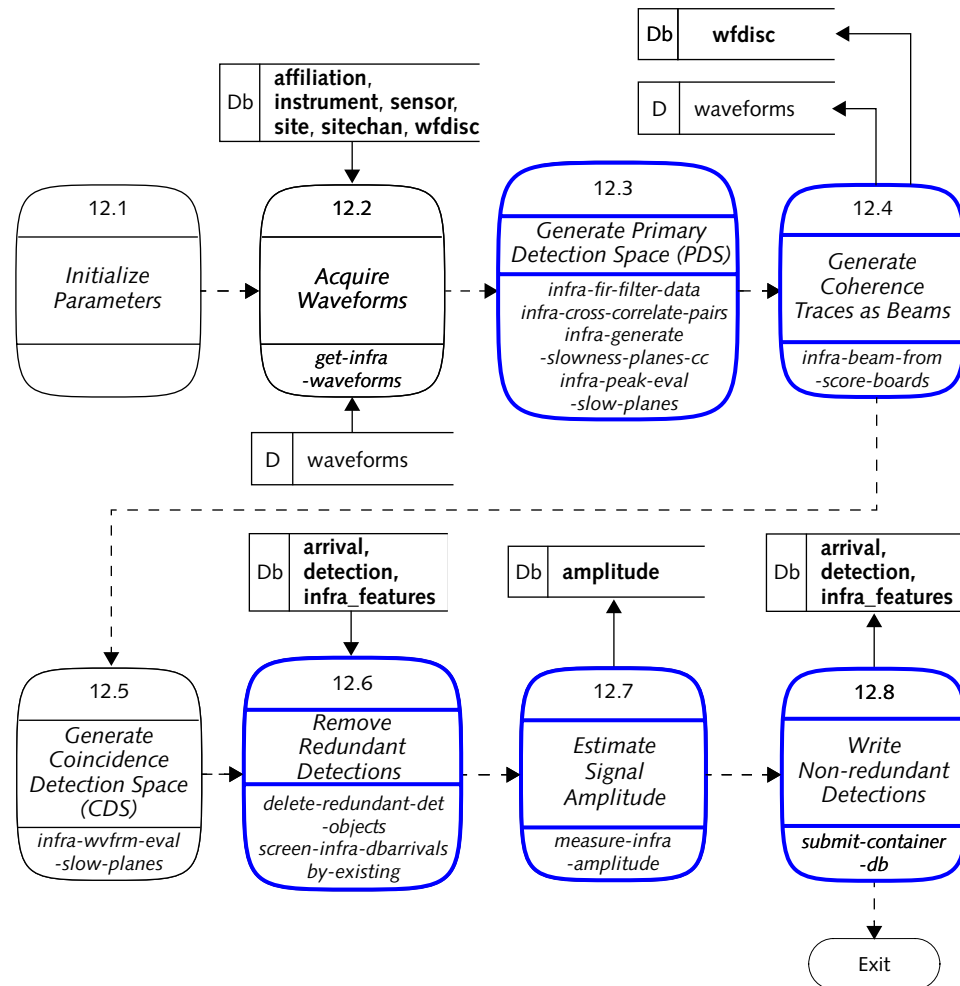


FIGURE 41. ARCHITECTURE OF INFRADET

## ▼ Infrasonic Processing

**TABLE 177: INPUT PARAMETERS FOR AUTOMATIC INFRASONIC DETECTION PROCESSING**

Par Name	Source	Default Value	Description
<i>sta</i>	command line	<none>	station code
<i>start-time</i>	command line	<none>	start-time of processing interval
<i>end-time</i>	command line	<none>	end-time of processing interval
<i>wfdisc-extension-time</i>	par file	<none>	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>qcstats-scheme-file</i>	par file	<none>	filename containing the QCStats source code
<i>affiliation-table</i>	par file	<b>affiliation</b>	table containing station and network affiliations
<i>site-table</i>	par file	<b>site</b>	table containing site-location information
<i>sitechann-table</i>	par file	<b>sitechann</b>	table containing station-channel information
<i>sensor-table</i>	par file	<b>sensor</b>	table containing specific calibration information for physical channels
<i>instrument-table</i>	par file	<b>instrument</b>	table containing generic (default) calibration information about a station
<i>in-wfdisc-table</i>	par file	<b>wfdisc</b>	input table containing waveform file header and descriptive information
<i>out-amplitude-table</i>	par file	<b>amplitude</b>	output table containing amplitude estimates generated by this application

**TABLE 177: INPUT PARAMETERS FOR AUTOMATIC INFRASONIC DETECTION PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>out-arrival-table</i>	par file	arrival	output table containing signal arrival information generated by this application
<i>out-detection-table</i>	par file	detection	output table containing signal detection information generated by this application
<i>out-infra_features-table</i>	par file	infra_features	output table containing signal attributes extracted by this application
<i>out-wfdisc-table</i>	par file	wfdisc	output table containing waveform file header and descriptive information generated by this application
<i>qcstats-table</i> <sup>1</sup>	par file	<none>	table containing statistical measures of the input waveforms developed by the quality-control function and written by this application
<i>perform-qcstats</i> <sup>1</sup>	par file	0 (false)	flag for generating and writing quality-control statistics: 0 means do not collect the data quality statistics

1. optional symbols local to *InfraDet*

Table 178 describes the output of *InfraDet*. *InfraDet* writes the results to the **amplitude**, **arrival**, **detection**, and **infra\_features** tables. (*InfraDet* also generates and writes coherence traces [as beams]. There are two output entities per coherence trace: a time-series file [with extension “.w”] and a corresponding record to the **wfdisc** table.)

## ▼ Infrasonic Processing

**TABLE 178: DATA PRODUCED BY AUTOMATIC INFRASONIC DETECTION PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>amplitude</b>	database table	signal amplitude for each infrasonic detection
<b>arrival</b>	database table	arrival attributes of each infrasonic detection
<b>detection</b>	database table	detection attributes of each infrasonic detection
<b>infra_features</b>	database table	waveform attributes, both time-domain and frequency-domain, for each infrasonic detection
<b>qcstats</b> <sup>1</sup>	database table	statistical attributes of the waveforms developed by the quality-control procedures
<b>wfdisc</b>	database table	auxiliary beams developed from data generated in the Primary Detection Space
".w"	binary files	coherence trace; time-series sample values

1. The table is written to only if appropriate processing parameter values have been set to allow for this.

The time-series for each coherence trace is written to the UNIX filesystem in the directory and filenames delineated by the *dir* and *dfile* attributes of the **wfdisc** table. The directory name is of the form */base/YYYY/JJJ/* where *JJJ* is the Julian day of the year (with leading zeros as required), and *YYYY* is the four-digit year for the given input waveforms used to develop the information generated in this processing interval. The base directory path is set in the *DFX-infra-detection.par* file by the *data-output-base-directory* parameter.

Table 241 on page 451 indicates the specific attributes written for each of these database tables. The duplication of information between the **arrival** and **detection** tables, and between these and the **amplitude** table is intentional; it supports parameter estimate refinement procedures.

## Processes

The following processes shown in Figure 41 on page 343 are described in this section:

- *Initialize Parameters [12.1]*
- *Acquire Waveforms [12.2]*
- *Generate Primary Detection Space (PDS) [12.3]*
- *Generate Coherence Traces as Beams [12.4]*
- *Generate Coincidence Detection Space (CDS) [12.5]*
- *Remove Redundant Detections [12.6]*
- *Estimate Signal Amplitude [12.7]*
- *Write Non-Redundant Detections [12.8]*

### Initialize Parameters [12.1]

*InfraDet* requires a variety of parameters for describing the station to be processed, for acquiring waveforms, and for controlling the processing performed. The *Initialize Sites* process acquires the relevant parameters and binds the Scheme symbols required for subsequent processing. This process is implemented by the global Scheme function `init-<name>-recipe`, where `<name>` is the recipe name. Table 179 lists the full set of recipes required for *InfraDet* processing.

## ▼ Infrasonic Processing

TABLE 179: RECIPE INFORMATION INITIALIZED FOR INFRADET PROCESSING

Recipe: Symbol: <Name>	General or Site-specific	Type	Description
<i>beam-recipe(s):</i> <i>beam-rec-con:</i> <i>beam</i>	site-specific	container	<i>beam-recipe</i> objects that define the extent and sampling of the vector-slowness coverage and filter bands
<i>infra-recipe:</i> <i>infra-rec-obj:</i> <i>infra</i>	site-specific	GObj	<i>infra-recipe</i> object parameters governing the infrasonic-detection processing
<i>QC-recipe:</i> <i>qc-rec-obj:</i> <i>qc</i>	site-specific	GObj	<i>qc-recipe</i> object parameters governing the quality-control processing
<i>time-interval-recipe:</i> <i>ti-rec-con:</i> <i>ti</i>	general	container	<i>time-interval-recipe</i> object parameters governing the definition of the applicable time-interval for amplitude-estimation processing
<i>amplitude-recipe:</i> <i>amp-rec-con:</i> <i>amplitude</i>	site-specific	container	<i>amplitude-recipe</i> object parameters governing the amplitude-estimation processing

The functions, `init-<name>-recipe`, return a recipe container or a recipe object (see Table 179). For all but the amplitude recipes these recipe initialization functions are called by Scheme language code such as:

```
(set! <name>-obj (init-<name>-recipe))
```

The functions have no arguments; the recipe parameters are obtained directly from the CVAR datastore.

The amplitude-recipe initialization function is slightly different and is invoked with the following Scheme language code:

```
(set! amprec-con (init-amplitude-recipe beam-rec-con
ti-rec-con))
```



The function requires two arguments, which are described in Table 180. The function returns a container of *amplitude-recipe* objects.

**TABLE 180: FUNCTIONAL INTERFACE OF INIT-AMPLITUDE-RECIPE**

Argument Name	Usage	Type	Description
<i>beam-recipe-con</i>	input	container	<i>beam-recipe</i> objects
<i>ti-rec-con</i>	input	container	<i>time-interval-recipe</i> objects
N/A	returned value	container	<i>amplitude-recipe</i> objects

### Acquire Waveforms [12.2]

This process seeks available waveforms for the given station and processing interval and reads these waveforms into memory. It also checks the quality of the data to identify problems and apply minor repairs to the acquired waveforms. This process is implemented by the local Scheme function `get-infra-waveforms`. It is called by the Scheme language code:

```
(set! wfdata-con (get-infra-waveforms infra-rec-obj
                                     beam-rec-con qc-rec-obj))
```

The function requires three arguments, which are described in Table 181.

**TABLE 181: FUNCTIONAL INTERFACE OF GET-INFRA-WAVEFORMS**

Argument Name	Usage	Type	Description
<i>infra-rec-obj</i>	input	GObj	<i>infrasonic detection processing recipe</i> object
<i>beam-rec-con</i>	input	container	<i>beam-recipe</i> objects
<i>qc-rec-obj</i>	input	GObj	<i>processing-recipe</i> object for checking the quality of the data
n/a	returned value	container	<i>wfdata</i> objects

▼ **Infrasonic Processing**

The standard method for acquiring waveforms is discussed in “Reading Waveforms” on page 58. *InfraDet* uses this method with the following principal differences:

- The requested data interval (symbol *data-interval-list*) is obtained from the *DFX/libinfra* function `get-infra-data-interval`.
- The actual data interval (symbol *processing-interval-list*) is obtained from the *DFX/libinfra* function `get-infra-processing-interval`.
- After the *wfdata* container (symbol *wfdata-con*) is filled, the number of objects (sensors) it contains is evaluated to determine whether a sufficient number (three or more) are present to allow for infrasonic processing.
- After the quality-control processing is called, the number of objects in the *wfdata* container is again evaluated to determine whether a sufficient number are still present. This second count is necessary because quality-control processing has the potential to delete the data from sensors if these data are found to be excessively defective. If the sensor count in the quality-checked *wfdata* container is less than three, the application exits; otherwise, it returns a quality-checked *wfdata* container to the calling application.

For this application, for the given station and processing time interval, this function performs the following activities:

1. Extract parameters from CVAR directly. Create initially null local symbols for *sta-list*, *chan-list*, *data-interval-list*, *wfdata-con*, and *wfmem-con*. Set the local symbol *perform-qcstats* in accordance with whether or not the CVAR datastore requires the production of quality-control statistical measures.
2. Obtain an initial container of waveforms objects (local symbol *wfdata-con*) by querying the database via the local Scheme function `get-initial-wfdata-con` (for general information on how database queries are implemented see “Querying the Database” on page 41).

3. Determine whether or not *wfdata-con* is null. If it is null, exit via the global Scheme function `error-fatal`, and return a formatted error message; otherwise, go to step 4.
4. Determine the requested data interval (symbol *data-interval-list*) via the *DFX/libinfra* function `get-infra-data-interval`. (This function accounts for edge effects from filter start-up and tail-off, lag-space requirements for beamforming, *stav* and *ltav* start-up lengths, and so on.)
5. Determine whether or not *data-interval-list* is null. If it is null, exit via the global Scheme function `error-fatal`, and return a formatted error message; otherwise, go to step 6.
6. Set the desired processing time interval to the values returned from `get-infra-data-interval`.
7. Extract the list of stations (symbol *sta-list*) and channels (symbol *chan-list*) from the initial waveforms container (symbol *wfdata-con*) via the intrinsic Scheme functions `set` and `unique`, the global Scheme function `map-container`, and the common *libgobj* Scheme function `extract-gobj-attr` (common *libgobj*), used in conjunction with the `lambda` construct.
8. Extract the list of unique station-channel pairs via the Scheme functions `set`, `unique-test`, `map-container`, `list`, and `extract-gobj-attr`, used in conjunction with the `lambda` construct.
9. Obtain a container of *wfmem* objects (symbol *wfmem-con*) from the **wfdisc** database table via the global Scheme function `query-for-wfmem-by-sta-chan-time`.
10. Determine whether or not *wfmem-con* is null. If it is null, exit via the generic Scheme function `fatal-error`, and return a formatted error message; otherwise, go to step 11.
11. Read the data from the disk into the *wfmem* objects via the *DFX/libdata* function `read-waveforms`. Return a container of *wfdata* objects that point at *wfmems* that contain time-series data for the given *sta-list*, *chan-list*, and desired processing time interval.

▼ **Infrasonic Processing**

12. Determine whether or not *wfdata-con* is null. If it is null, exit via the generic Scheme function `fatal-error`, and return a formatted error message; otherwise, go to step 13.
13. For this returned container of waveforms objects (*wfdata-con*), determine the processing time interval (symbol *processing-interval-list*) via the *DFX/libinfra* function `get-infra-processing-interval`.
14. Determine whether or not *processing-interval-list* is null. If it is null, exit via the generic Scheme function `fatal-error`, and return a formatted error message; otherwise, go to step 15.
15. Set the start-times and end-times in the CVAR datastore via the intrinsic Scheme functions `car` and `cadr`, and the common *libscheme* Scheme function `set-cvar`.
16. Determine whether sufficient (three or more) sensors are present in the waveforms container to allow for infrasonic processing. If they are not, issue a formatted warning message, and return control to the calling program via the intrinsic Scheme function `exit`; otherwise, go to step 17.
17. Check and repair, as possible, the waveforms via the *DFX/libqc* function `qc-waveforms-all`.
18. If quality-control statistical measures are required, use the *DFX/libqc* functions `qc-perform-stats` and `process-qcstats`.
19. Delete bad station-channels from the waveforms container returned by the `qcstats` functions,

**Generate Primary Detection  
Space (PDS) [12.3]**

This application bases the declaration of infrasonic detection on coincidence (that is, time overlapped) in the occurrence of energy in excess of a fixed threshold and in the occurrence of a level of spatial coherence above a fixed threshold. Both the energy excess and the level of spatial coherence are estimated for beams steered to the same value of vector-slowness. To determine the vector-slowness values in which to efficiently search for coincidence detection, this application first generates

a Primary Detection Space (PDS) based on spatial coherence alone. The candidate detections found in this PDS are used to define the vector-slowness values for which coincidence detection is sought. This process generates candidate detections based upon the level of spatial coherence within an array of infrasonic sensors. The level of spatial coherence, also known as the detection statistic, is estimated from the mean of the aligned normalized coherent cross-correlations among the band-pass-filtered waveforms received at the sensor array. The alignments are selected to span a range of vector-slowness values for the given site, sensor subgroup, and spectral band as defined by the beam recipes. The set of detection-statistic values that are calculated for each of the vector-slowness hypotheses completes the generation of a slowness plane (that is, a three-dimensional entity in which the spatial coherence level [one dimension] is given as a function of two orthogonal components of vector-slowness [two more dimensions], azimuth, and magnitude-slowness). The PDS is generated by taking the slowness planes for all of the filter bands, all of the processing time windows (also referred to as epochs), and all of the subgroups of three or more sensors at the given array station. Candidate detections are developed by comparing the detection statistic with a threshold value for each slowness plane in the PDS. In each slowness plane, up to six candidate detections have their vector-slowness coordinates refined, provided they are all above the threshold and are sufficiently separated in slowness from one another. The default limit of six candidate detections is set in the *DFX libinfra* via a C-code compiler directive of the `#define` type. These candidate detections are the basis for further processing to achieve declared detections.

This process is implemented by the following *DFX/libinfra* functions:

```
infra-fir-filter-data  
infra-cross-correlate-pairs  
infra-generate-slowness-planes-cc  
infra-peak-eval-slow-planes
```

These functions are invoked by Scheme language calls in the following sequence:

```
(set! bpf-con-con (infra-fir-filter-data wfdata-con  
beam-rec-con infra-rec-obj))
```

▼ **Infrasonic Processing**

```
(set! ncc-con-con (infra-cross-correlate-pairs
  bpf-con-con beam-rec-con infra-rec-obj))

(set! slplcc-con-con (infra-generate-slowness-planes-cc
  ncc-con-con beam-rec-con infra-rec-obj))

(set! sbcc-con-con (infra-peak-eval-slow-planes
  slplcc-con-con ncc-con-con beam-rec-con infra-rec-obj))
```

The functions require the arguments described in Table 182.

**TABLE 182: FUNCTIONAL INTERFACE OF FUNCTIONS USED TO GENERATE PRIMARY DETECTION SPACE (PDS)**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>beam-rec-con</i>	input	container	<i>beam-recipe</i> objects
<i>infra-rec-obj</i>	input	GObj	<i>infrasonic detection processing recipe</i> object
<i>bpf-con-con</i>	input and output	container of containers	<i>bandpass-filtered waveform</i> objects (actually, traditional <i>beam</i> objects in structure)
<i>ncc-con-con</i>	input and output	container of containers	<i>normalized cross-correlation</i> objects
<i>slplcc-con-con</i>	input and output	container of containers	<i>slowness-plane</i> objects (generated from aligned normalized cross-correlations)
<i>sbcc-con-con</i>	output	container of containers	<i>score-board</i> objects (generated from slowness planes)
<i>bpf-con-con</i> <i>ncc-con-con</i> <i>slplcc-con-con</i> <i>sbcc-con-con</i>	returned values	container of containers	container of the given class

The function `infra-fir-filter-data` produces bandpass-filtered waveforms from the quality-checked waveforms input by `wfdata-con`. The bandpass filters are finite impulse response (FIR) filters, which are characterized by phase-responses that are linear with frequency. Thus, the group-delay is constant with respect to frequency. Filters of this nature preserve coherence from station to station within a sensor array. The returned output from this function is a container of containers of *beam* objects (symbol *bpf-con-con*). Each filter band has an inner container. This inner container contains one *beam* object for each sensor. The data vectors in these *beam* objects only contain the FIR bandpass-filtered waveforms for a single sensor.

The function `infra-cross-correlate-pairs` produces normalized cross-correlations between the bandpass-filtered waveforms from the independent pairs of sensors. The sensor pairs are defined using the beam-recipe element groups for each filter band. The generation of normalized cross-correlation functions is performed for only those sensor pairs required for a given filter band. The returned output from this function is a container of *normalized cross-correlation* containers (symbol *ncc-con-con*). Each distinct filter band has an inner container. This inner container contains all of the *normalized cross-correlation* objects, one for each sensor pair for each time position of the processing window.

The function `infra-generate-slowness-planes-cc` combines all the normalized cross-correlation pairs to generate a detection statistic as a function of azimuth and magnitude-slowness. The generation of a slowness plane populates a six-dimensional space: three explicit dimensions, time (processing window, also known as epochs), frequency (filter band), and space (beam group), and three intrinsic dimensions within each slowness plane, azimuth ( $\theta$ ), magnitude-slowness, ( $|s|$ ), and detection-statistic value ( $[snr]_e$ ). The PDS is the collection of slowness planes for the given band and beam group combinations for all the successive epochs in the overall processing interval. The PDS is treated as a three-dimensional space where the fundamental entity is the slowness plane where each slowness plane is itself a three-dimensional entity. The output from this function is a container of *slowness-plane* containers (symbol *slplcc-con-con*). Each distinct filter band has an inner container. This inner container contains all of the slowness planes, one for each beam group for each time position of the processing window.

▼ **Infrasonic Processing**

The function `infra-peak-eval-slow-planes` evaluates each of the slowness planes to determine whether or not there is evidence of a candidate detection. The evaluation is based upon the level of spatial coherence exhibited. A *score-board* object is generated for each *slowness-plane* object. The *score-board* object is a tabulation of the candidate detection's attributes and the single best refined peak regardless of whether it is a detection. The attributes include the detection-statistic value, the azimuth, the magnitude trace-velocity (inverse of magnitude-slowness), and the error estimates for each of the two vector-slowness coordinates. For each *slowness-plane* object a *score-board* object is returned by this function; thus, the returned entity is a container of *score-board* containers (symbol *sbcc-con-con*). Each distinct filter band has an inner container. This inner container contains all of the score boards, one for each beam group for each time position of the processing window.

For more information on this function see the "Detailed Description of Generate Primary Detection Space (PDS) [12.3]" on page 366.

**Generate Coherence Traces as Beams [12.4]**

This process produces infrasonic detection-related time series that can be displayed alongside waveforms in ARS. These time series are formatted identically as traditional beams and are referred to as coherence traces because they are derived from the spatial coherence information that was generated to form the PDS. These coherence traces allow the infrasonic detection results to be related to the waveforms from which the PDS was generated. The following coherence traces are extracted from the PDS:

- time series of maximum spatial coherence (transformed to the F-statistic)
- time series of the azimuth at which this maximum level of spatial coherence was realized
- time series of the magnitude-trace-velocity at which this maximum level of spatial coherence was realized



These time series, when viewed alongside the array channel waveforms, allow an analyst to relate detections with coherent features in the waveforms.

This process is implemented by the *DFX/libinfra* function `infra-beam-from-score-boards`, which is called by the Scheme language code:

```
(set! y-beam (infra-beam-from-score-boards sbcc-con-con
      "type" "name" "beam_id"))
```

The function requires four arguments, which are described in Table 183.

**TABLE 183: FUNCTIONAL INTERFACE OF INFRA-BEAM-FROM-SCORE-BOARDS**

Argument Name	Usage	Type	Description
<i>sbcc-con-con</i>	input	container of containers	<i>score-board</i> objects
<i>type</i>	input	character-string	indicating either max or avg
<i>name</i>	input	character-string	name of parameter (in the <i>score-board</i> objects) to be turned into beam
<i>beam_id</i>	input	character-string	this beam's designation in displays such as <i>ARS</i>
n/a	returned value	GObj	<i>beam</i> object

The function `infra-beam-from-score-boards` reads the score boards and extracts a numerical attribute specified by the *name* parameter. The time series of this parameter is transformed into a *beam* object, which is provided as the returned value. The beams are written to the UNIX filesystem with corresponding records written to the **wfdisc** database table.

*InfraDet* creates and writes three "max"-types of beams as a standard set of coherence traces. Beams of type "max" are time-series of the given parameter (identified by *name*) for the specific combination of processing band and beam-group

▼ **Infrasonic Processing**

that has the highest level of spatial coherence. The specific combination of band and beam-group may vary from one time sample to the next. Table 184 describes the attributes of these beams. For more information on this function see “Detailed Description of Generate Coherence Traces as Beams [12.4]” on page 376.

**TABLE 184: AUXILIARY BEAMS CREATED AND WRITTEN BY  
AUTOMATIC INFRASONIC DETECTION**

Beam_id	Parameter <sup>1</sup>
"mx"	$f\text{-stat} = 1 + \text{"used\_channels"} * \text{"tst\_stat"}$ $= 1 + J * (\text{snr})_e$ <p>where <math>J</math> = number of non-collocated sensors</p>
"az"	"az_mean" azimuth in degrees
"sl"	"v_tr_mean" magnitude-trace-velocity in meters/second

1. The parameter entities in quotes are the actual names of the parameter in the *score-board* object or selected parameters derived from score-board parameters.

### **Generate Coincidence Detection Space (CDS) [12.5]**

Based on the candidate detections generated in the PDS, this process develops declared detections that exhibit both a sufficient level of spatial coherence and a sufficient level of energy in excess of the ambient noise level. This process is implemented by the *DFX/libinfra* function `infra-wvfrm-eval-slow-planes`. This function is called by the Scheme language code:

```
(set! sbcc-con-con (infra-wvfrm-eval-slow-planes
  sbcc-con-con ncc-con-con beam-rec-con infra-rec-obj))
```

The function requires four arguments, which are described in Table 185.

**TABLE 185: FUNCTIONAL INTERFACE OF FUNCTIONS USED TO GENERATE COINCIDENCE DETECTION SPACE (CDS)**

Argument	Usage	Type	Description
<i>sbcc-con-con</i>	input	container of containers	<i>score-board</i> objects
<i>ncc-con-con</i>	input	container of containers	<i>normalized cross-correlation</i> objects
<i>beam-rec-con</i>	input	container	<i>beam-recipe</i> objects
<i>infra-rec-obj</i>	input	GObj	<i>infrasonic detection processing recipe</i> object
n/a	returned value	container of containers	<i>infrasonic detection</i> objects

The function `infra-wvfrm-eval-slow-planes` evaluates the score boards to generate a list of all candidate detections from the PDS. It then groups the detections that are similar in their vector-slowness coordinates into clusters. These clusters are then used to form beam pairs steered to the vector-slowness of the cluster. The pair is comprised of an energy beam and an F-statistic beam. For each cluster's beam pair, the energy beam is evaluated for energy in excess of the ambient noise level, and the F-statistic beam is evaluated for a sufficiently high spatial coherence level. For each cluster, a detection is declared when both beams satisfy detection criteria for some period of time in common. A "vernier" f-k spectrum is generated to quantify the level of spatial coherence in precisely the direction of the incident plane wave. It also refines the plane wave's arrival parameters, azimuth, and slowness. In addition, error estimates are generated for the vector-slowness coordinates and both time-domain and frequency-domain signal features are estimated. The function `infra-wvfrm-eval-slow-planes` returns a container of containers of *infra-detection* objects. Each distinct cluster that was found to contain one or more declared detections has an inner container. This inner container holds all of the *infra-detection* objects from a single cluster ordered by arrival time. All of the inner containers are held within the outer container (symbol *infra-det-con-con*).

## ▼ Infrasonic Processing

The coincidence detection processing algorithm is as follows:

1. The first subprocess groups the PDS candidate detections. The best (in a spatial-coherence-level sense) candidate detection among all the score boards is used as the “seed” contact<sup>1</sup> of the first cluster. All other candidate detections in the entire PDS whose vector-slowness values are sufficiently close to the vector-slowness value of this “seed” contact are assigned to this first cluster. After it is assigned to a contact cluster, a candidate detection is no longer available as a seed and cannot be assigned to any other cluster. After the first contact cluster has been created, the best candidate detection remaining unassigned is used as the “seed” contact for the second contact cluster. All unassigned candidate detections whose vector-slowness values are sufficiently close to this “seed” contact are assigned to this second cluster. This process is repeated until all candidate detections have been assigned to a cluster. A cluster may have only a single candidate detection. Generally between one and a few dozen clusters are formed for a typical processing interval. The vector-slowness “closeness” criterion used in forming the contact clusters is set in the infrasonic detection processing recipe in the parameter *slowness\_criterion*. The smaller this parameter’s value, the more contact clusters can be formed.
2. The second subprocess generates the pairs of beams (the energy beam and the F-statistic beam) that are steered in the vector-slowness direction of each cluster’s seed contact. The energy beam is a traditional delay-and-sum coherent beamformed from the bandpass-filtered waveforms. The F-statistic beam is a sample-by-sample estimate of the F-statistic, where the integration time (the processing window duration) is defined by the *coherent-integration-time* parameter in the infrasonic detection processing recipe.
3. The third subprocess evaluates the beam pairs. For each cluster the energy beam-detection statistic is calculated by computing the ratio of the STA to the time-lagged LTA of the energy beam. Both the STA and

---

1. The terms “candidate detection” and “contact” are used interchangeably.

the LTA use an L1 norm. (See [IDC5.2.1] for more information on computing STA and LTA.) The time series of the STA/LTA is searched for candidate energy peaks. The biggest peak is the first energy peak defined, and it is retained if its ratio is above the threshold value set in the infrasonic detection processing recipe in parameter *"sta\_lta\_threshold"*. If this peak is below the threshold, there are no energy-peak detections and therefore no coincidences and no declared detections. If more than one energy peak is above the threshold, then the energy beam is searched for the next energy peak that occurs sufficiently before or after the first peak. If there is such an above-threshold energy peak, then it is considered for retention. It is retained if there is a "valley" between energy peaks that is sufficiently deep compared with the STA/LTA value of the previously retained peak. The valley-depth criterion, expressed as a fraction of the previously retained peak STA/LTA value and the time-closeness criterion for retaining two successive energy peaks, is set in C-code within *DFX/libinfra*. This procedure is repeated until all candidate energy peaks have been considered. The time duration of the energy packet associated with each retained energy peak is defined by the times immediately before and after the given peak where the STA/LTA value drops below the threshold value. Each retained peak defines an energy packet of finite time-duration and peak time.

A coincidence detection is declared wherever an energy packet has one or more time samples of overlap with an above threshold level on the F-statistic beam. The threshold applied to the F-statistic beam is set in the infrasonic detection processing recipe in the parameter *coherent\_threshold*.

4. The fourth subprocess generates a "vernier" f-k spectrum for each coincidence detection in a given cluster. The f-k spectrum is centered at the time of the energy packet's peak. The waveform-analysis interval is set to match the time duration of this energy packet. The filtering of the waveforms and the spectral extent of the f-k spectrum are set identically and to the band limits of the processing band that was used to generate the seed contact for the cluster being processed. In contrast with the FIR

▼ **Infrasonic Processing**

bandpass filtering applied to the waveforms used in the PDS, the waveforms for the “vernier” f-k spectrum are filtered with Infinite Impulse Response (IIR) bandpass filters; that is, the recursive (*libfilter*) algorithms are used. (See [IDC5.2.1] for more information on filtering waveforms.) This f-k spectrum is “vernier” in the sense that the waveforms are pre-aligned in time to a vector-slowness close to that of the seed contact for the cluster being processed, and the slowness plane is very finely resolved in vector-slowness covering only a small slowness area. The global peak value of F-statistic quantifies the spatial coherence of this detection.

5. The fifth subprocess estimates several of the signal's attributes for each coincidence detection in a given cluster. These estimates include:
  - errors in azimuth and magnitude-slowness (square root of the sum of estimated measurement and modeling variances)
  - corner frequency and its error
  - zero-crossing frequency and its error
  - start-time, time-duration, and error in time-duration of the energy packet
  - start-time, time-duration, and error in time-duration of the interval of above threshold spatial-coherence
  - start-time, time-duration, and error in time-duration of the interval of coincidence
  - total energy in the energy packet
  - level of coherent snr
  - coherent period

For each coincidence detection in a given cluster, an *infra-detection* object is created (a detection is declared). All of the *infra-detection* objects for a given cluster are put into a container. Each of these containers are put in an outer container. The outer container of *infra-detection* containers (symbol *infra-det-con-con*) is returned by this function.

### Remove Redundant Detections [12.6]

This process creates the appropriate entities with which to update the **arrival**, **detection**, and **infra\_features** tables of the database. These entities are created from the information in the container of containers of *infra-detection* objects. This process performs two levels of redundancy elimination. First, for a given cluster, it eliminates redundant detections that are too close in time to one another. Second, it eliminates the detections too close in time and vector-slowness with detections already in the database. If a new detection is eliminated (deleted), then all associated entities (the *dbarrival*, the *dbdetection*, and the *dbinfra\_features*) are also deleted.

This process prints a warning message for each deleted detection. It also prints an error message if all detections developed during this processing time-interval for this site have been deleted. This function generally follows the conventions described in “Screening Database Containers” on page 43; however, for more information on this function see “Detailed Description of Remove Redundant Detections [12.6]” on page 381.

### Estimate Signal Amplitude [12.7]

*InfraDet* estimates the waveform amplitude of each declared detection in a fashion appropriate for airborne infrasonic energy. *Estimate Signal Amplitude* measures the maximum peak-to-trough amplitude across a single zero-crossing of the “beam-formed” bandpass-filtered waveforms within a defined processing window. The qualifier *beamformed* is in quotes because the beam formation is strictly a matter of data formatting to allow existing library functions to be applied; there is no sum over the waveforms from multiple sensors. Rather, the waveforms from each sensor are measured, and an evaluation process is applied to select that sensor’s measurement that is to be reported.

This process first checks if required instrument response files are accessible via the local Scheme function `instrument-response-files-exist?`. This function is called by the Scheme language code:

## ▼ Infrasonic Processing

```
(set! files-exist (instrument-response-files-exist?
  wfdata-con))
```

The function requires one argument, which is described in Table 186.

**TABLE 186: FUNCTIONAL INTERFACE OF INSTRUMENT-RESPONSE-FILES-EXIST?**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
n/a	returned value	logical scalar	1 if the files exist 0 if the files do not exist

If one or more of the required instrument-response files are not accessible, then a warning message is printed, and this function returns a value of zero; otherwise, it returns a value of unity.

The process *Estimate Signal Amplitude* estimates the waveform amplitude in a manner appropriate for infrasonic processing. The fundamental estimation algorithm is a traditional seismic amplitude estimation function, *measure-amplitude*, which is provided in the library *DFX/libamp*.

This process is implemented by the local Scheme function *measure-infra-amplitude* as well as by a number of other local, global, and intrinsic Scheme functions. This function is called by the Scheme language code:

```
(set! dbamp-out-con (measure-infra-amplitude
  infra-det-con-con wfdata-con amp-rec-con infra-rec-obj))
```

The function requires four arguments, which are described in Table 187.



TABLE 187: FUNCTIONAL INTERFACE OF MEASURE-INFRA-AMPLITUDE

Argument Name	Usage	Type	Description
<i>infra-det-con-con</i>	input	container of containers	<i>infra-detection</i> objects
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>amp-rec-con</i>	input	container	<i>amplitude estimation processing recipe</i> objects
<i>infra-rec-obj</i>	input	GObj	<i>infrasonic detection processing recipe</i> object
n/a	returned value	container	<i>dbamplitude</i> objects

This function operates on the container of *infra-detection* containers, an *amplitude recipe* container, a *wfdata* container, and an *infrasonic detection processing recipe* object. It returns a *dbamplitude* container. A warning message is printed if this function fails to estimate the signal amplitude for any of the *infra-detection* objects.

If the *amp* and *per* attributes of infrasonic detections in the **arrival** table of the database are to be updated with the amplitude and period estimates from the **amplitude** table where arrival identification numbers (*arids*) match, then the local symbol *update-arrivals* must be reset to unity. Currently, the **arrival** table is not being updated, and a warning message is being returned. If this updating is switched on by resetting the symbol *update-arrivals* to unity, then the local Scheme function *update-arrivals-with-amp-info* will adjust the *dbarrival* objects in the container of non-redundant detections to have their *amp* and *per* parameters set in accordance with the *amp* and *per* parameters in the corresponding *dbamp* objects.

For more information about this function see “Detailed Description of Estimate Signal Amplitude [12.7]” on page 391.

▼ **Infrasonic Processing****Write Non-Redundant Detections [12.8]**

This process writes the *dbarrival*, *dbdetection*, *dbinfra\_features*, and *dbamplitude* containers to the appropriate database tables. With the exception of the *dbamplitude* container, the other containers are submitted in the following order: *dbarrivals*, *dbdetections*, and *dbinfra\_features*. The *dbamplitudes* are only submitted to the database if the local symbol *do-amp* is set to unity. For more information about submitting results to the database see “Writing to the Database” on page 44.

After the non-redundant detections have been written to the database successfully, the database is “committed” by the global Scheme function `commit-db`.

For more information about this process see “Detailed Description of Write Non-Redundant Detections [12.8]” on page 399 .

**Detailed Description of Generate Primary Detection Space (PDS) [12.3]**

*InfraDet* declares detections by seeking coincidence of energy in excess of a fixed threshold and spatial coherence level in excess of another fixed threshold for a pair of beams steered in the vector-slowness of an hypothesized infrasound source. It would be computationally burdensome to search all feasible directions and magnitude-slowness values, in the required small step sizes, with the coincidence detector directly. Rather, the PDS is generated to provide a comparatively small number of vector-slowness values that are worthy of further detection search efforts. The process *Generate Primary Detection Space (PDS)* accomplishes this. Figure 42 shows the architecture of this process, which invokes four *DFX/libinfra* subprocesses. The first subprocess generates bandpass-filtered waveforms. Then next subprocess generates normalized cross-correlation functions from the bandpass-filtered waveforms via the function `infra-cross-correlate-pairs`. The next subprocess generates slowness planes, and, finally, the last subprocess evaluates the slowness planes and summarizes their information content into score boards: one score board for each slowness plane.

Four new object classes are defined in *DFX/libinfra*: *bandpass-filter-specification* objects, *normalized cross-correlation* objects, *slowness-plane* objects, and *score-board* objects. This process has the container of *beam* objects (which hold the bandpass-filtered waveforms) as well as these four new object classes as its output.

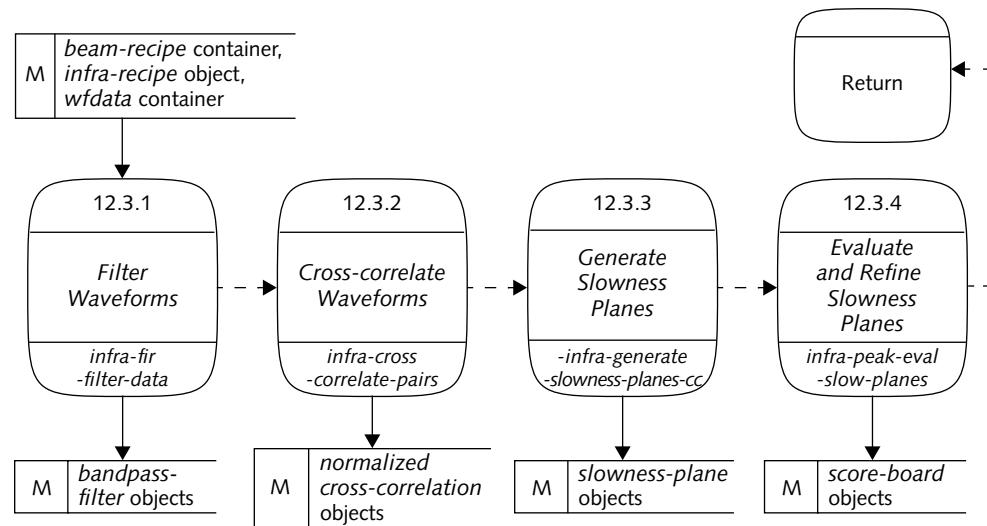


FIGURE 42. ARCHITECTURE OF GENERATE PRIMARY DETECTION SPACE (PDS)

The following subprocesses shown in Figure 42 are described in this section:

- *Filter Waveforms [12.3.1]*
- *Cross-correlate Waveforms [12.3.2]*
- *Generate Slowness Planes [12.3.3]*
- *Evaluate and Refine Slowness Planes [12.3.4]*

▼ **Infrasonic Processing****Filter Waveforms [12.3.1]**

This subprocess applies finite impulse response (FIR) bandpass filters (BPFs) to the quality-checked waveforms using filter parameters in the beam recipes or each sensor referenced in the *beam-recipe* container. This subprocess is implemented by the *DFX/libinfra* function *infra-fir-filter-data*. This function is called by Scheme language code such as:

```
(set! bpf-con-con (infra-fir-filter-data wfdata-con
    beam-rec-con infra-rec-obj))
```

The function requires three arguments, which are described in Table 188.

**TABLE 188: FUNCTIONAL INTERFACE OF INFRA-FIR-FILTER-DATA**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>beam-rec-con</i>	input	container	<i>beam-recipe</i> objects
<i>infra-rec-obj</i>	input	GObj	<i>infrasonic detection processing recipe</i> object
n/a	returned value	container of containers	<i>beam</i> objects

The FIR filters are selected from a set of filters contained in a file specified by the infrasonic detection processing recipe parameter *filter-file-name-infra*. The filters that match closest to frequency filter limits specified in the beam recipes (symbol *beam-rec-con*) are used. Beam recipes that are associated with the same FIR filter are thereby processed in the same filter band. After the processing band allocations have been made, each of the FIR filters is applied to every one of the sensors available in the waveforms container (symbol *wfdata-con*) regardless of whether that sensor is ever to be processed in that particular band.

FIR filters are used to ensure that three performance goals are achieved: (i) there is no frequency-dependent group delay in the filtered waveforms, (ii) the in-band amplitude response is flat to within +/- 0.3 dB, has unity gain, and the highest out-

of-band response is at least 50 dB down, and (iii) the transition regions from in-band to stop-band are monotonically decreasing and less than 20 mHz wide at a sample rate of 20 samples per second. These attributes are important for preserving phase coherence and for avoiding contamination from nearby strong and spatially coherent spectral features.

The output from this function is a container of containers of *beam* objects. Each distinct spectral processing band has an inner container. This inner container holds all of the *beam* objects, one for each sensor. The data vector referenced by these *beam* objects contains only the FIR bandpass-filtered waveforms for a single sensor; all other vectors referenced by the *beam* object for the given sensor are null. All of the inner containers, one for each distinct processing band, are held within the returned outer container (symbol *bpf-con-con*).

### Cross-correlate Waveforms [12.3.2]

This subprocess generates sequences of normalized cross-correlation functions from the FIR bandpass-filtered waveforms. This subprocess is implemented by the *DFX/libinfra* function *infra-cross-correlate-pairs*. This function is called by Scheme language code such as:

```
(set! ncc-con-con (infra-cross-correlate-pairs
  bpf-con-con beam-rec-con infra-rec-obj))
```

The function requires three arguments, which are described in Table 189.

**TABLE 189: FUNCTIONAL INTERFACE OF INFRA-CROSS-CORRELATE-PAIRS**

Argument Name	Usage	Type	Description
<i>bpf-con-con</i>	input	container of containers	<i>beam</i> objects
<i>beam-rec-con</i>	input	container	<i>beam-recipe</i> objects
<i>infra-rec-obj</i>	input	GObj	<i>infrasonic detection processing recipe</i> object
n/a	returned value	container of containers	<i>normalized cross-correlation</i> objects

▼ **Infrasonic Processing**

The determination of which sensor pairs are to be cross-correlated for each spectral processing band is based upon an evaluation of the beam groups and beam recipes in the *beam-recipe* container (symbol *beam-rec-con*). The processing-window time-duration and the update-time interval between successive cross-correlation processing windows is governed by the *infrasonic detection processing recipe* object (symbol *infra-rec-obj*). The lag extent of each normalized cross-correlation function is the same for every sensor pair in every processing band for a given *beam-recipe* container (symbol *beam-rec-con*), *bandpass-filtered waveforms* container, and *infrasonic detection processing recipe* object. This lag extent is defined by the longest baseline separation distance among the sensors in the waveform container, the maximum magnitude-slowness that can be generated by the beam recipes, and the slowness-refinement extent required in the infrasonic detection processing recipe. The output from this function is a container of *normalized cross-correlation* containers (symbol *ncc-con-con*). Each distinct filter band has an inner container. Within this inner container are all of the normalized *cross-correlation* objects, one for each sensor pair for each time position (epoch) of the processing window. Within each inner container the sequence of *normalized cross-correlation* objects is as follows:

```

first epoch:    first pair, second pair, third pair, ..., p-th pair;
second epoch:  first pair, second pair, third pair, ..., p-th pair;
               ...;
n-th epoch:    first pair, second pair, third pair, ..., p-th pair.
```

All of the inner containers are held within the returned outer container.

### **Generate Slowness Planes [12.3.3]**

This subprocess generates sequences of slowness planes by combining all the normalized cross-correlation pairs for a given processing band, beam group, and processing-time window. This subprocess is implemented by the *DFX/libinfra* function *infra-generate-slowness-planes-cc*. This function is called by Scheme language code such as:

```
(set! slplcc-con-con (infra-generate-slowness-planes-cc
  ncc-con-con beam-rec-con infra-rec-obj))
```

The function requires three arguments, which are described in Table 190.

**TABLE 190: FUNCTIONAL INTERFACE OF INFRA-GENERATE-SLOWNESS-PLANES-CC**

Argument Name	Usage	Type	Description
<i>ncc-con-con</i>	input	container of containers	<i>normalized cross-correlation</i> objects
<i>beam-rec-con</i>	input	container	<i>beam-recipe</i> objects
<i>infra-rec-obj</i>	input	GObj	<i>infrasonic detection processing recipe</i> class
n/a	returned value	container of containers	<i>slowness-plane</i> objects

Each slowness plane is sampled in azimuth and magnitude-slowness in accordance with the beam recipes (symbol *beam-rec-con*). Each beam recipe specifies an azimuth and magnitude-slowness that define a two-dimensional vector-slowness value. For a given vector-slowness, the arrival time of a plane wave at each sensor is defined precisely. The method of combining the normalized cross-correlation functions into the detection statistic is defined by the integer-valued combination-criterion parameter within the infrasonic detection processing recipe (symbol *infra-rec-obj*). The default value for this parameter specifies the arithmetic mean of the time-aligned normalized cross-correlation functions. This detection statistic is converted to an equivalent signal-to-noise ratio by the fixed transformation

$$(\text{snr})_e = ncc / (1 - ncc)$$

where *ncc* is the normalized cross-correlation value (a real-valued number between -1 and 1). The foregoing transformation only is applied for *ncc* values zero or larger and, in fact, values smaller than approximately 0.2 to 0.3 are of no interest to infrasonic signal processing applied to airborne signals from distant sources. Thus,

## ▼ Infrasonic Processing

the default arithmetic mean of time-aligned normalized cross-correlation function populates the slowness plane with the spatial-coherence detection statistic,  $(\text{snr})_e$ , at each hypothesized sample of vector-slowness (azimuth, magnitude-slowness).

For a typical processing interval (a few tens of minutes in total duration) the normalized cross-correlation functions are generated for dozens to hundreds of successive epochs. For each processing epoch, a slowness plane is generated for each combination of filter band and beam group required in the *beam-recipe* container. Thus, the generation of a slowness-plane populates a six-dimensional space; three explicit dimensions, time (processing epochs), frequency (filter band), and space (beam-group). And there are three intrinsic dimensions within each slowness plane, azimuth, magnitude-slowness, and detection-statistic value,  $(\text{snr})_e$ , at that azimuth and magnitude-slowness (for that “group,” in that “band,” over that “epoch”).

The PDS is the collection of slowness planes for the given filter-band and beam-group combinations for all the successive epochs in the overall processing interval. The PDS is treated as a three-dimensional space where the fundamental entity is the slowness plane (in reality, each slowness plane is itself a three-dimensional entity). The output from this function is a container of *slowness-plane* containers (symbol *slplcc-con-con*). Each distinct filter band has an inner container. Within this inner container are all of the *slowness-plane* objects for a given band, one for each beam-group for each epoch. Within each inner container the sequence of *slowness-plane* objects is as follows:

```

first epoch:  first beam-group, second beam-group,
               third-beam group, ..., g-th beam-group;

second epoch: first beam-group, second beam-group,
               third beam-group, ..., g-th beam-group;
               ...;

n-th epoch:   first beam-group, second beam-group,
               third beam-group, ..., g-th beam-group.
```

All of the inner containers are held within the returned outer container.



### Evaluate and Refine Slowness Planes [12.3.4]

This subprocess evaluates each of the slowness planes to determine if there is evidence of detection of an infrasonic plane wave based upon the level of spatial coherence exhibited. This subprocess is implemented by the *DFX/libinfra* function *infra-peak-eval-slow-planes*. This function is called by Scheme language code such as:

```
(set! sbcc-con-con (infra-peak-eval-slow-planes
  splcc-con-con ncc-con-con beam-rec-con infra-rec-obj))
```

The function requires four arguments, which are described in Table 191.

**TABLE 191: FUNCTIONAL INTERFACE OF INFRA-PEAK-EVAL-SLOW-PLANES**

Argument Name	Usage	Type	Description
<i>splcc-con-con</i>	input	container of containers	<i>slowness-plane</i> objects
<i>ncc-con-con</i>	input	container of containers	<i>normalized cross-correlation</i> objects
<i>beam-rec-con</i>	input	container	<i>beam-recipe</i> objects
<i>infra-rec-obj</i>	input	GObj	<i>infrasonic detection processing recipe</i> object
n/a	returned value	container of containers	<i>score-board</i> objects

Each slowness plane for a given processing epoch, band, and beam-group is processed and evaluated on its own merits without regard to other bands or beam-groups and without regard to preceding or following epochs. For each slowness plane a *score-board* object (symbol *sbcc-con-con*) is created. A score board is a concise summary of a slowness plane. The processing algorithm is as follows:

## ▼ Infrasonic Processing

1. The first step of this subprocess is to develop candidate peaks in each slowness plane. A picket in the slowness plane is characterized by three coordinates: the level of spatial coherence and the azimuth and magnitude-slowness on the slowness plane. Each picket of the slowness plane that is not on the edge of the slowness plane is compared with its four nearest neighbors. Only those pickets whose value of  $(\text{snr})_e$  (the detection statistic) exceeds all four nearest neighbors are candidates for further processing; these are candidate peaks.
2. The second step of this subprocess is to compare the peaks to the local threshold value. If there are candidate peaks, each one is compared with the threshold value of the *beam-recipe* object that produced that peak's picket; specifically, a fixed fraction of the snr value in the beam recipe for that (azimuth, magnitude-slowness). The default value of the fixed fraction is 0.7. Even if no candidate peak passes the threshold test, the single best (highest value of  $(\text{snr})_e$ ) peak is still accepted for further processing.
3. The third step of this subprocess is to sort the list of candidate peaks by the  $(\text{snr})_e$  value.
4. The fourth step of this subprocess is to reject candidate peaks that are too close in slowness to one another. That is, the candidate peak with the best value of  $(\text{snr})_e$  is accepted for further processing, then the list is searched for the next best candidate peak whose slowness distance from those candidate peaks already accepted for further processing is greater than a specified criterion. Candidate peaks deemed too close to an accepted candidate peak are dropped from further consideration. This process is complete when each candidate peak originally on the list has either been retained for further processing or dropped because it was too close in slowness to a candidate peak of higher  $(\text{snr})_e$  value.
5. The fifth step of this subprocess is to generate refinement slowness planes. Each candidate peak on this list is refined. That is, the vector-slowness coordinates of the peak constitute the center of a highly sampled (in vector-slowness) slowness plane of limited slowness extent. Within this refined slowness plane the vector-slowness coordinates of the picket with locally maximum value of  $(\text{snr})_e$  are sought. The detection

statistic is also better quantified because the value of vector-slowness used in the time-alignment of normalized cross-correlation functions is better matched to those of the incident plane wave. In addition, the refinement slowness plane allows the widths of the maximal peak to be estimated in both azimuth and magnitude-slowness directions. The refined peaks are sorted with respect to the refined  $(\text{snr})_e$  values. Up to six refined peaks, those with the highest  $(\text{snr})_e$  values above their snr thresholds, are output to a *score-board* object. Even if no refined peak exceeds its snr threshold, information about the highest refined peak is included in the *score-board* object for that slowness plane. The score board contains a field indicating the number of candidate detections output (one for each above-threshold refined peak). In addition, it contains the vector-slowness coordinates, the candidate peaks' widths, and  $(\text{snr})_e$  value of each candidate peak, as well as some other measures.

Because the *slowness-plane* objects and *score-board* objects have a one-to-one correspondence, a container of *score-board* containers is returned by *infra-peak-eval-slow-planes*. Each distinct filter-band has an inner container. Within this inner container are all of the *score-board* objects from that filter band, one for each beam-group for each epoch. The sequence of *score-board* objects within each inner container is as follows:

```

first epoch:  first beam-group, second beam-group,
               third beam-group, ..., g-th beam-group;

second epoch: first beam-group, second beam-group,
               third-beam group, ..., g-th beam-group;
               ...;

n-th epoch:   first beam-group, second beam-group,
               third beam-group, ..., g-th beam-group.

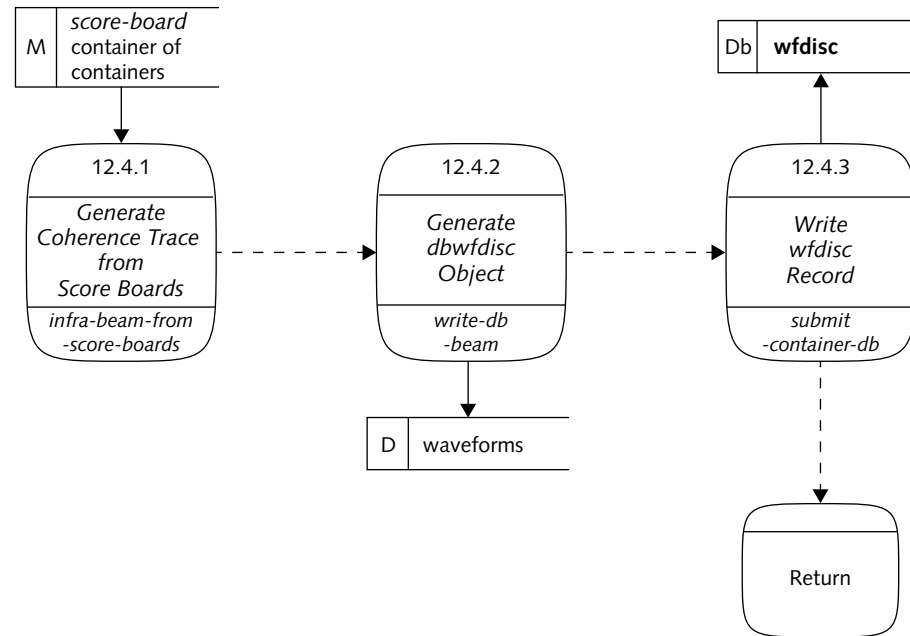
```

All of the inner containers are held within the returned outer container.

▼ **Infrasonic Processing****Detailed Description of Generate Coherence Traces as Beams [12.4]**

In IDC operations the results of the *InfraDet* application are reviewed by human analysts who examine the waveforms (via *ARS*) in the vicinity of actual detections as well as in time intervals where a detection should exist if an hypothesized event were detected on the given sensor. To provide the analyst with some mechanism for knowing whether infrasonic signals were detectable at a given infrasonic array station, a time series of the level of spatial coherence at the given station is made available in the form of a beam, referred to as a coherence trace, which can be displayed alongside the waveforms for the given station. Where this coherence trace exhibits sufficiently high levels, infrasonic plane waves may have been present. In addition to the level of spatial coherence, two additional traces are available. These exhibit a time series of the azimuth of the source and a time series of the magnitude of the trace velocity of the plane wave. (During intervals of genuine infrasonic signal reception, the values in these latter coherence traces tend to stabilize at the azimuth of the source and the inverse magnitude-slowness of the source.)

The process *Generate Coherence Traces as Beams* generates and stores these coherence traces. Figure 43 shows the architecture of the process, which consists of three subprocesses: (i) generate a coherence trace from score boards, (ii) generate a *dbbeam* object, and (iii) write a *wfdisc* record, including indicative information to the **wfdisc** database table and a binary “.w” file of time-series samples to the UNIX filesystem. These subprocesses are implemented by one *DFX/libinfra* function, one *DFX/libio* function, and a global Scheme function. The process *Generate Coherence Traces as Beams* has as its output a record to the **wfdisc** database table and a corresponding “.w” file for each of the three coherence traces generated for each processing time interval and station.



**FIGURE 43. ARCHITECTURE OF GENERATE COHERENCE TRACES AS BEAMS**

The following subprocesses shown in Figure 43 are described in this section:

- *Generate Coherence Trace from Score Boards [12.4.1]*
- *Generate dbwfdisc Object [12.4.2]*
- *Write wfdisc Record [12.4.3]*

#### **Generate Coherence Trace from Score Boards [12.4.1]**

This subprocess generates a time series of selected parameters extracted from the score boards, which were produced while generating the PDS. Each time series is reformatted as a *beam* object, which can be displayed by ARS. This subprocess is implemented by the *DFX/libinfra* function `infra-beam-from-score-boards`. This function is called by a Scheme language code such as:

## ▼ Infrasonic Processing

```
(set! y-beam (infra-beam-from-score-boards sbcc-con-con
      type name beam_id))
```

The function requires four arguments, which are described in Table 192.

**TABLE 192: FUNCTIONAL INTERFACE OF INFRA-BEAM-FROM-SCORE-BOARDS**

Argument Name	Usage	Type	Description
<i>sbcc-con-con</i>	input	container of containers	<i>score-board</i> objects
<i>type</i>	input	character-string	type of selection where more than a single datum exists per time sample: either <i>max</i> or <i>avg</i>
<i>name</i>	input	character-string	name of the parameter in the <i>score-board</i> objects to be turned into a beam: for example, <i>az_mean</i> , <i>f_stat</i> , ...
<i>beam_id</i>	input	character-string	this beam's designation in displays such as <i>ARS</i>
n/a	returned value	GObj	<i>beam</i> object

This function reads the container of *score-board* containers and, for each epoch, forms a real-valued scalar from a numerical attribute specified by the character-string *name* in the *score-board* object. The time series of this real-valued scalar is transformed into a *beam* object. The channel name in the *beam* object is the character-string *beam\_id*. Where the number of processing bands/beam-groups yields more than a single *score-board* object for each epoch, more than a single datum exists that can be attributed to the coherence trace at each time sample. For each given epoch, two mechanisms are available for transforming the specified numerical attribute among the set of *score-board* objects into a single real-valued scalar datum. One method selects the value of the parameter from that *score-board* object with the highest (snr)<sub>e</sub>. This is specified by setting the symbol *type* to

max (see Table 192). The other method uses the weighted average,  $(\text{snr})_e$ , being the weight) of the specified parameter among the set of score boards available for each processing epoch. In this case symbol *type* is set to *avg* (see Table 192).

The function *infra-beam-from-score-boards* returns a *beam* object.

### Generate *dbwfdisc* Object [12.4.2]

This subprocess writes the time series of beam data as a binary file to the UNIX file-system in the appropriate directory with the appropriate filename with “.w” extension. It also generates a *dbwfdisc* object with appropriate information about the beam including the assignment of a unique waveform identification number. This subprocess is implemented by the *DFX/libio* function *write-db-beam*. This function is called by a Scheme language code such as:

```
(set! db-y-wfdisc (write-db-beam db-conn-obj y-beam wfid))
```

The function requires three arguments, which are described in Table 193.

**TABLE 193: FUNCTIONAL INTERFACE OF WRITE-DB-BEAM**

Argument Name	Usage	Type	Description
<i>db-conn-obj</i>	input	GObj	<i>database-connection</i> object
<i>y-beam</i>	input	GObj	<i>beam</i> object
<i>wfid</i>	input	integer	unique waveform identification number for this coherence trace
n/a	returned value	GObj	<i>dbwfdisc</i> object

The *dbbeam* object (symbol *db-y-beam*) is put into a *dbbeam* container by the global Scheme function *insert-container*. The global Scheme functions *say-db-connection* and *query-for-lastid* are used to supply the *database-connection* object and the unique waveform-identification number. This function returns a beam-associated *dbwfdisc* object to be submitted to the *wfdisc* database table.

## ▼ Infrasonic Processing

**Write wfdisc Record [12.4.3]**

This subprocess submits the *dbwfdisc* object produced in the *Generate dbwfdisc Object [12.4.2]* process to the **wfdisc** database table for output. The subprocess is implemented by three global Scheme functions: `create-keyed-list-container`, `insert-container`, and `submit-container-db`. The last of these functions is the most significant and is called by Scheme language code such as:

```
(submit-container-db db-y-wfdisc-con table-name)
```

The function requires two arguments, which are described in Table 194.

**TABLE 194: FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB (WFDISC)**

Argument Name	Usage	Type	Description
<i>db-y-wfdisc-con</i>	input	container	<i>dbwfdisc</i> objects
<i>table-name</i>	input	character-string	database table name where header information about this coherence trace is to be written
n/a	returned value	logical	indicates success (t) or failure (nil)

This subprocess first creates a *dbwfdisc* container via the global Scheme function `create-keyed-list-container`. It then inserts the *dbwfdisc* object into the *dbwfdisc* container via the global Scheme function `insert-container`. Finally, it writes the *dbwfdisc* container (symbol *db-y-wfdisc-con*) to the database to the table whose name is given by the symbol *table-name*. The local Scheme function `depart-with-grace` is used to return a warning message if the container *db-y-wfdisc-con* to be submitted to the database is null or if the submission process fails. Where there is a failure, the function `depart-with-grace` also rolls back the database to its state at the time of its last commit.



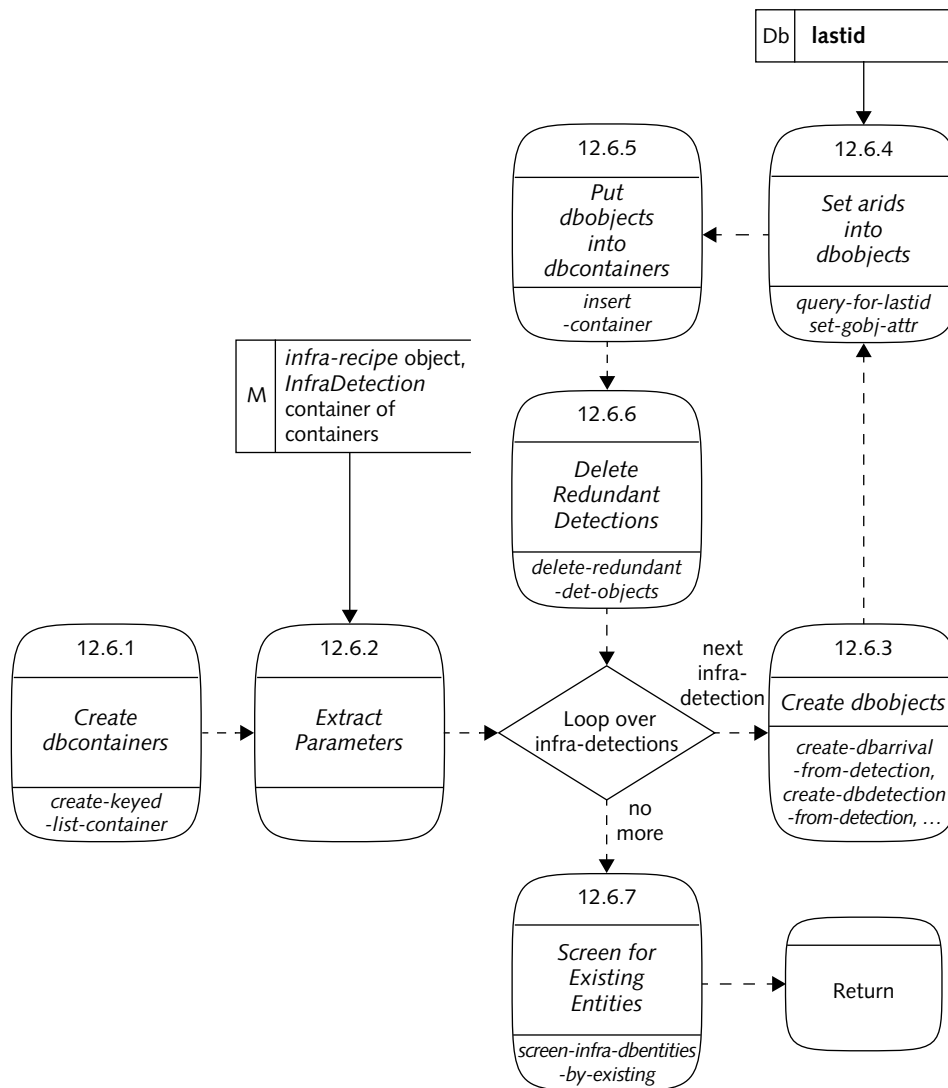
## Detailed Description of Remove Redundant Detections [12.6]

*InfraDet* has the potential for generating detections for each processing interval and station for which it is invoked. Although *InfraDet* is typically invoked for a sequence of adjacent "abutting" processing intervals, whose nominal start- and end-times are advanced by a fixed amount equal to the time duration of any given interval, several factors require that the waveforms acquired and processing performed in any given processing interval start earlier than the nominal start-time and run later than the nominal end-time. Thus, successive processing intervals are overlapped in time. Because of this overlap, the same signal may be detected at the end (in the overlap region) of a given processing interval and re-detected in the start (in the overlap or even the nominal processing region) of the next processing interval. Downstream processing assumes that there will be only one record in the **arrival** table for a physical arrival. Consequently, *InfraDet* performs processing to eliminate both the redundant detections developed within a given single processing interval as well as detections that are redundant relative to previous processing intervals.

The process *Remove Redundant Detections* accomplishes this. Figure 44 shows the architecture of the process. First, it passes through two subprocesses that create dbcontainers and extract parameters. Next it enters a loop over the newly generated infrasonic detections. This loop consists of four subprocesses, which create dbobjects, set arids in dbobjects, put dbobjects into dbcontainers, and delete redundant detections, respectively. Finally, the process screens for existing entities (*dbarrival* as well as corresponding *dbdetection* and *dbinfra\_features* objects).

The following subprocesses shown in Figure 44 are described in this section:

- *Create dbcontainers* [12.6.1]
- *Extract Parameters* [12.6.2]
- *Create dbobjects* [12.6.3]
- *Set arids into dbobjects* [12.6.4]
- *Put dbobjects into dbcontainers* [12.6.5]
- *Delete Redundant Detections* [12.6.6]

▼ **Infrasonic Processing**■ *Screen for Existing Entities [12.6.7]***FIGURE 44. ARCHITECTURE OF REMOVE REDUNDANT DETECTIONS**

**Create dbcontainers [12.6.1]**

This subprocess creates containers to hold the *dbarrival*, *dbdetection*, and *dbinfra\_features* objects created from each *infra-detection* object produced by the coincidence detector. It is implemented by inline Scheme code. The relevant global Scheme functions include:

```
create-keyed-list-container
say-db<XXX>-object-class
```

as well as the *DFX/libutil* Scheme function:

```
make-gobj-name
```

These functions are applied to create the dbcontainers listed in Table 195.

**TABLE 195: DBCONTAINERS CREATED BY CREATE DBCONTAINERS**

Symbol	Description
<i>dbarrival-con</i>	container of <i>dbarrival</i> objects
<i>dbdetection-con</i>	container of <i>dbdetection</i> objects
<i>dbinfra_features-con</i>	container of <i>dbinfra_features</i> objects

This subprocess uses `make-gobj-name` and `say-db<XXX>-object-class` to provide unique character-string names for the given container and to provide a definition of the object class (that is symbolized as `db<XXX>`) that the container is intended to hold, respectively.

**Extract Parameters [12.6.2]**

This subprocess extracts the parameters listed in Table 196 from the CVAR datastore and from the infrasonic detection processing recipe.

▼ **Infrasonic Processing****TABLE 196: PARAMETERS EXTRACTED BY EXTRACT PARAMETERS**

Symbol	Source	Type	Description
<i>sta</i>	CVAR	character-string	station code
<i>start-time</i>	CVAR	real	start-time of current processing interval
<i>end-time</i>	CVAR	real	end-time of current processing interval
<i>amplitude-table</i>	CVAR	character-string	name of <b>amplitude</b> table in data-base
<i>arrival-table</i>	CVAR	character-string	name of <b>arrival</b> table
<i>detection-table</i>	CVAR	character-string	name of <b>detection</b> table
<i>infra_features-table</i>	CVAR	character-string	name of <b>infra_features</b> table
<i>azi-limit</i>	<i>infra-rec-obj</i>	real	limit for matching azimuths
<i>slow-limit</i>	<i>infra-rec-obj</i>	real	limit for matching magnitude-slowness
<i>redundancy-time-limit</i>	<i>infra-rec-obj</i>	real	limit for matching arrival time

The three limit parameters, *azi-limit*, *slow-limit*, and *redundancy-time-limit*, are each set to 50% of the *azimuth-criterion*, 50% of the *slowness-criterion*, and 50% of the *coherent-integration-time* parameters held in the infrasonic detection processing recipe.

**Create dbobjects [12.6.3]**

Within the loop over *infra-detection* objects, this subprocess uses three *DFX/libinfra* functions and two *DFX/libdb* functions to create dbobjects of the *dbarrival*, *dbdetection*, and *dbinfra\_features* classes from each *infra-detection* object. The three *DFX/libinfra* functions are:

```
create-GObjDet-from-GObjInfraDet  
create-GObjDbInfra_Features-from-GObjInfra_Feat  
create-GObjInfra_Feat-from-GObjInfraDet
```

The two *DFX/libdb* functions are

```
create-dbarrival-from-detection  
create-db detection-from-detection
```

First, a *detection* object is extracted from the *infra-detection* object via `create-GObjDet-from-GObjInfraDet`. Then a *dbarrival* object and a *dbdetection* object are extracted from the *detection* object. Finally, an *infra\_features* object is extracted from the *infra-detection* object via `create-GObjInfra_Feat-from-GObjInfraDet`. This *infra\_features* object is transformed to a *dbinfra\_features* object via `create-GObjDbInfra_Features-from-GObjInfra_Feat`. The fields in a *dbarrival* object correspond to the fields in an **arrival** table; the fields in a *dbdetection* object correspond to the fields in a **detection** table; and the fields in a *dbinfra\_features* object correspond to the fields in an **infra\_features** table (see [IDC5.1.1Rev2]).

#### Set arids into dbobjects [12.6.4]

Within the loop over *infra-detection* objects, this subprocess puts the same unique arrival identification number (symbol *arid*) into each of the *dbarrival*, *dbdetection*, and *dbinfra\_features* objects that were derived from a given single *infra-detection* object. It uses the global Scheme function `query-for-lastid` and the inline common *libgobj* Scheme function `set-gobj-attr`. In addition, it sets this same *arid* value into the *infra-detection* object that was used to derive the three dbobjects.

#### Put dbobjects into dbcontainers [12.6.5]

Within the loop over *infra-detection* objects, this subprocess uses the global Scheme function `insert-container` to insert each of the three types of dbobjects derived from each *infra-detection* object into the corresponding class of dbcontainers.

▼ **Infrasonic Processing****Delete Redundant Detections [12.6.6]**

Within the loop over *infra-detection* objects, this subprocess deletes infrasonic detections that are closer in time to one another than the allowed value of *redundancy-time-limit*. This subprocess is implemented by the local Scheme function `delete-redundant-det-objects`. This function is called by Scheme language code such as:

```
(set! out-infra-det-con (delete-redundant-det-objects
  infra-det-con redundancy-time-limit))
```

The function requires two arguments, which are described in Table 197.

**TABLE 197: FUNCTIONAL INTERFACE OF DELETE-REDUNDANT-DET-OBJECTS**

Argument Name	Usage	Type	Description
<i>infra-det-con</i>	input and output	container	<i>infra-detection</i> objects
<i>redundancy-time-limit</i>	input	real	time difference limit to consider two successive detections redundant
n/a	returned value	container	<i>infra-detection</i> objects

Within the loop over *infra-detection* objects, this subprocess compares the arrival times of successive detections. If two successive detections are closer in time than the time criterion (symbol *redundancy-time-limit*) then the detection with the larger value of coherent snr (that is,  $[snr]_e$ ) is retained, and the other detection is deleted from the container. The function `delete-redundant-det-objects` returns a container of *infra-detection* objects that has no two successive detections closer in arrival time than the time criterion. The redundant *infra-detection* objects are deleted from the input container as well.

**Screen for Existing Entities [12.6.7]**

This subprocess screens the *dbarrival*, *dbdetection*, and *dbinfra\_features* objects for each infrasonic detection for the presence of a sufficiently similar infrasonic detection already in the database. To do so, it first screens the *dbarrival* objects for sufficiently similar arrivals in the **arrival** table. Those *dbarrival* objects deemed redundant are deleted from the *dbarrival* container, and the corresponding input *infra-detection* object has its *arid* attribute set to the integer value -1. This allows for simple handling of the amplitude estimation processing, which is discussed in the “Detailed Description of Estimate Signal Amplitude [12.7]” on page 391, specifically the subsection “Extract Parameters [12.7.1]” on page 393. This subprocess is implemented by three local Scheme functions:

```
screen-infra-dbarrivals-by-existing
screen-infra-dbdetects-by-existing
screen-infra-dbinfra_features-by-existing
```

The first function is called by Scheme language code such as:

```
(set! out-dbarrival-con
  (screen-infra-dbarrivals-by-existing sta start-time
    end-time redundancy-time-limit azi-limit slow-limit
    in-dbarrival-con infra-det-con-con))
```

The function requires eight arguments, which are described in Table 198.

**TABLE 198: FUNCTIONAL INTERFACE OF SCREEN-INFRA-DBARRIVALS-BY-EXISTING**

Argument Name	Usage	Type	Description
<i>sta</i>	input	character-string	station code of station currently being processed
<i>start-time</i>	input	real	start-time of current processing interval
<i>end-time</i>	input	real	end-time of current processing interval

## ▼ Infrasonic Processing

**TABLE 198: FUNCTIONAL INTERFACE OF SCREEN-INFRA-DBARRIVALS-BY-EXISTING (CONTINUED)**

Argument Name	Usage	Type	Description
<i>redundancy-time-limit</i>	input	real	time difference limit to consider two successive detections redundant
<i>azi-limit</i>	input	real	azimuthal difference limit to consider two successive detections redundant
<i>slow-limit</i>	input	real	magnitude-slowness difference limit to consider two successive detections redundant
<i>in-dbarrival-con</i>	input and output	container	<i>dbarrival</i> objects
<i>infra-det-con-con</i>	input and output	container of containers	<i>infra-detection</i> objects
n/a	returned value	container	<i>dbarrival</i> objects

The function `screen-infra-dbarrivals-by-existing` performs a query to the `arrival` table to obtain arrivals within the processing interval, adjusted for extra waveform data at each end of the processing interval, for this station. It then loops over the list of database arrivals. For each arrival found in the database it loops over the newly generated arrivals in the input *dbarrival* container. It compares the arrival times, azimuths, and magnitude-slowness values for each database arrival with each newly generated arrival. If the following three conditions are met,

- the arrival times are within the time criterion (symbol *redundancy-time-limit*)
- the azimuths are within the azimuth criterion (symbol *azi-limit*)
- the magnitude-slownesses are within the magnitude-slowness criterion (symbol *slow-limit*)



then the newly generated *dbarrival* object is deleted from the input *dbarrival* container, and the *arid* attribute in the matching input *infra-detection* object is set to the integer value -1.

The function `screen-infra-dbarrivals-by-existing` returns a *dbarrival* container whose contents have no matches in the **arrival** database table for the given station and processing time interval. For more information see "Database Operations" on page 40.

After the *dbarrival* container has been screened for existing arrivals, the subprocess *Screen for Existing Entities* eliminates the appropriate *dbdetection* and *dbinfra\_features* objects from the *dbdetection* container and the *dbinfra\_features* container, respectively. Two local Scheme functions are used to accomplish this:

```
screen-infra-dbdetects-by-existing
screen-infra-dbinfra_features-by-existing
```

These two functions are applied in the same fashion. The function `screen-infra-dbdetects-by-existing` is called by a Scheme language code such as:

```
(set! out-dbdetect-con
  (screen-infra-dbdetects-by-existing in-dbarrival-con
    in-dbdetect-con))
```

The function requires two arguments, which are described in Table 199.

**TABLE 199: FUNCTIONAL INTERFACE OF SCREEN-INFRA-DBDETECTS-BY-EXISTING**

Argument Name	Usage	Type	Description
<i>in-dbarrival-con</i>	input	container	<i>dbarrival</i> objects
<i>in-dbdetect-con</i>	input and output	container	<i>dbdetection</i> objects
n/a	returned value	container	<i>dbdetection</i> objects

▼ **Infrasonic Processing**

This function loops over the *dbdetection* container. For each *dbdetection* object it loops over the *dbarrival* container (which has had the *dbarrival* objects corresponding to existing arrivals deleted from it). If the *arid* parameter of the *dbdetection* object does not match the *arid* parameter of any *dbarrival* object, then the *dbdetection* object is deleted from the *dbdetection* container. The function `screen-infra-dbdetects-by-existing` returns a *dbdetection* container that has no matches in the **detection** table.

The function `screen-infra-dbinfra_features-by-existing` is called by Scheme language code such as:

```
(set! in-dbinfra_features-con
  (screen-infra-dbinfra_features-by-existing
    in-dbarrival-con in-dbinfra_features-con))
```

The function requires two arguments, which are described in Table 200.

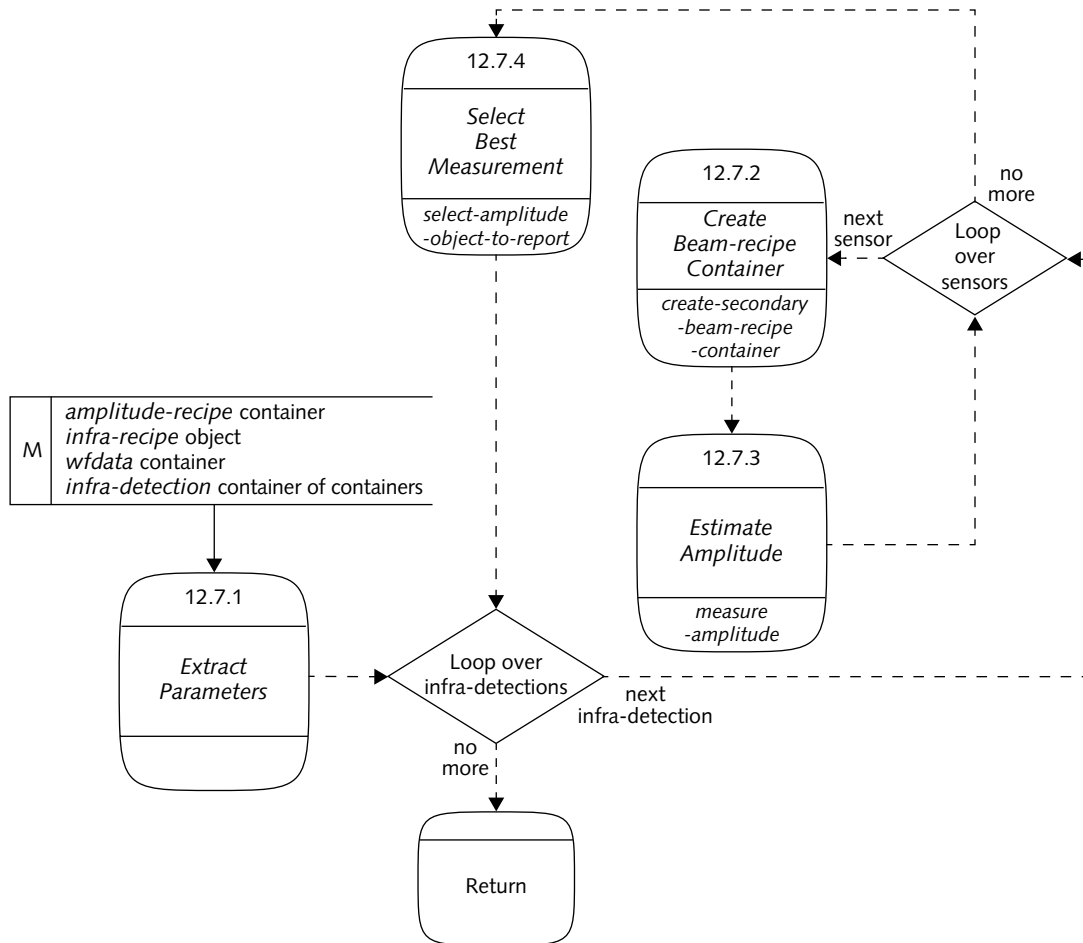
**TABLE 200: FUNCTIONAL INTERFACE OF SCREEN-INFRA-DBINFRA\_FEATURES-BY-EXISTING**

Argument Name	Usage	Type	Description
<i>in-dbarrival-con</i>	input	container	<i>dbarrival</i> objects
<i>in-dbinfra_features-con</i>	input and output	container	<i>dbinfra_features</i> objects
n/a	returned value	container	<i>dbinfra_features</i> objects

This function loops over the *dbinfra\_features* container. For each *dbinfra\_features* object it loops over the *dbarrival* container. If the *arid* parameter of the *dbinfra\_features* object does not match the *arid* parameter of any *dbarrival* object, then the *dbinfra\_features* object is deleted from the *dbinfra\_features* container. The function `screen-infra-dbinfra_features-by-existing` returns a *dbinfra\_features* container whose contents have no matches in the **infra\_features** table.

### Detailed Description of Estimate Signal Amplitude [12.7]

*InfraDet* produces estimates of the waveform amplitude in the vicinity of each declared detection. It uses the same amplitude measurement library functions employed in *SeisDet*. However, in contrast with the seismic approach where the estimation is applied to beamformed data, the infrasonic approach is applied to each sensor's waveforms on an individual basis. The reported amplitude is based on a selection of the most representative measurement from among the individual sensor results. The process *Estimate Amplitude*, implemented via the local Scheme function `measure-infra-amplitude`, is used to perform the waveform amplitude estimation for *InfraDet*. Figure 45 shows the architecture of the process. First it extracts parameters. Then it enters a nested pair of loops over the infrasonic detections (outer loop) and over the individual sensors (inner loop) used to develop this detection at this infrasonic station. Within the inner loop, for each individual sensor, a suitable single sensor beam recipe is generated and the amplitude estimation is performed. These activities are accomplished via the subprocesses *Create Beam-recipe Container* and *Estimate Amplitude*. After the inner loop has run over all relevant sensors and an amplitude estimate has been generated for each sensor, the subprocess *Select Best Measurement* (within the outer loop) is invoked. This subprocess evaluates the set of amplitude estimates and selects the most representative estimate to be attributed to the correct infrasonic detection. The subprocess *Select Best Measurement* then inserts this measurement (in the form of a *dbamplitude* object) into a *dbamplitude* container. When the all non-redundant infrasonic detections have been processed, the *dbamplitude* container is returned.

▼ **Infrasonic Processing****FIGURE 45. ARCHITECTURE OF ESTIMATE SIGNAL AMPLITUDE**

The following subprocesses shown in Figure 45 are described in this section:

- *Extract Parameters [12.7.1]*
- *Create Beam-recipe Container [12.7.2]*
- *Estimate Amplitude [12.7.3]*
- *Select Best Measurement [12.7.4]*

**Extract Parameters [12.7.1]**

This subprocess acquires the parameters required for amplitude estimation processing. It creates a container to hold the *dbamp* objects created from each *infra-detection* object by the coincidence detector. It is implemented by inline Scheme code. The relevant global Scheme functions include:

```
create-keyed-list-container
make-gobj-name
say-db<XXX>-object-class
```

In addition, the name (symbol *amplitude-table*) of the output **amplitude** table is extracted from the CVAR datastore. The local symbol *time-error* is set to the sta-time parameter value from the infrasonic detection processing recipe (symbol *infra-rec-obj*). Finally, the local symbol *time-window-factor* is set to unity.

The nested pair of loops is implemented as three nested loops. The outer loop over detections shown in Figure 45 is actually two nested loops: the first is a loop over the inner containers of *infra-detection* objects; the second (interior to the first) is a loop over the *infra-detection* objects themselves. The inner-most loop is the loop over sensors used to generate each given *infra-detection* object, as shown in Figure 45.

Between the loop over *infra-detection* objects and the inner loop over sensors, a variety of parameters are extracted for the current *infra-detection* object. The extracted parameters include the symbols described in Table 201.

**TABLE 201: PARAMETERS EXTRACTED AND SYMBOLS SET FOR MEASURE-INFRA-AMPLITUDE**

Symbol	Source	Type	Description
<i>time</i>	<i>infra-detection</i> object	real	detection time
<i>arid</i>	<i>infra-detection</i> object	integer	unique arrival identification number
<i>az-d-o</i>	<i>infra-detection</i> object	real	azimuth (degrees with respect to North)

## ▼ Infrasonic Processing

**TABLE 201: PARAMETERS EXTRACTED AND SYMBOLS SET FOR MEASURE-INFRA-AMPLITUDE (CONTINUED)**

Symbol	Source	Type	Description
<i>sl-d-o</i>	<i>infra-detection</i> object	real	magnitude-slowness (second/kilometer)
<i>ts-d-o</i>	local computation from <i>time</i> , <i>time-error</i> , and <i>time-window-factor</i>	real	start-time for window within which to perform amplitude estimation
<i>te-d-o</i>	local computation from <i>time</i> , <i>time-error</i> , and <i>time-window-factor</i>	real	end-time for window within which to perform amplitude estimation
<i>best-bmrec-obj</i>	<i>infra-detection</i> object	GObj	<i>beam-recipe</i> object
<i>bmrec-bmgrp-con</i>	local computation from <i>best-bmrec-obj</i>	container	<i>beam-element</i> objects
<i>num-sensors</i>	local computation from <i>bmrec-bmgrp-con</i>	real	number of sensors for current infrasonic detection

These local symbols, most of which are unique in value to the current *infra-detection* object, are used to create *beam-recipe* container(s) suitable for the subsequent infrasonic waveform amplitude estimation processing. If the *arid* value for the current *detection* object is equal to  $-1$ , then the given detection was found to be redundant and amplitude processing is not performed on this detection.

**Create Beam-recipe Container [12.7.2]**

Amplitude estimation in *DFX* requires the input waveforms to be structured and conveyed as a *beam* object. For infrasound, the use of beamformed data is not desired. Rather, the single-sensor instrument-response corrected waveforms (time-aligned to correspond to the detection vector-slowness, and bandpass filtered to the detection band) are the data on which to estimate the signal's amplitude. The amplitude estimate is the frequency-response corrected (to account for the bandpass filtering) arithmetic mean over the single-sensor measurements having matching measurement times.

The most efficient way to cast the input waveforms as a *beam* object is to perform the beamforming processing albeit with one major adjustment: the beam recipe contains a beam group with only a single sensor. Then, the beamforming returns the waveforms from that sensor, bandpass filtered as required, and slightly shifted in time as required by the beamsteering. By applying this technique for each sensor at the infrasonic array station, the measurements can be accomplished on a single sensor basis. Thus, the objective is to create a beam recipe for each sensor.

The subprocess *Create Beam-recipe Container* exists within the innermost loop over sensors. It creates a secondary *beam-recipe* container consisting of a beam group, which has only the current sensor and a coherent-beam recipe steered in the vector-slowness of the current detection. This beam recipe also has the filter parameters set to those values used to obtain the current detection. The single-sensor *beam-element* object is created by the local Scheme function `create-secondary-beam-element-object`. This function is called by Scheme language code such as:

```
(set! sec-bmelem-obj
  (create-secondary-beam-element-object best-bmrec-obj
    ith-sensor))
```

The function requires two arguments, which are described in Table 202.

**TABLE 202: FUNCTIONAL INTERFACE OF CREATE-SECONDARY-BEAM-ELEMENT-OBJECT**

Argument Name	Usage	Type	Description
<i>best-bmrec-obj</i>	input	GObj	<i>beam-recipe</i> object
<i>ith-sensor</i>	input	integer	integer index of current sensor
n/a	returned value	GObj	<i>beam-element</i> object

This function extracts the *beam-element* container from the input best *beam-recipe* object (symbol *best-bmrec-obj*) and then extracts the *beam-element* object for the *ith-sensor* (symbol *ith-sensor*) from this container. The function `create-secondary-beam-element-object` returns this single-sensor secondary *beam-element*

▼ **Infrasonic Processing**

object. Next, the common *libgobj* Scheme function `extract-gobj-attr` and the intrinsic Scheme function `set!` are used to create a beam-name (symbol *sec-beam-name*) from the station name of the single sensor in the secondary *beam-element* object. Finally, the local Scheme function `create-secondary-beam-recipe-container` is applied to form the special *beam-recipe* container that has the single-sensor *beam-element* object for this current sensor. This function is called by Scheme language code such as:

```
(set! sec-bmrec-con
  (create-secondary-beam-recipe-container best-bmrec-obj
    sec-bmelem-obj sec-beam-name))
```

The function requires three arguments, which are described in Table 203.

**TABLE 203: FUNCTIONAL INTERFACE OF CREATE-SECONDARY-BEAM-RECIPE-CONTAINER**

Argument Name	Usage	Type	Description
<i>best-bmrec-obj</i>	input	GObj	<i>beam-recipe</i> object
<i>sec-bmelem-obj</i>	input	GObj	<i>beam-element</i> object for the current single sensor
<i>sec-beam-name</i>	input	character-string	code of current single sensor to be used as the beam-name
n/a	returned value	container	<i>beam-recipe</i> objects

This function constructs a secondary *beam-recipe* container (symbol *sec-bmrec-con*) consisting of this best beam recipe and a *beam-element* container that contains only this single-sensor *beam-element* object (symbol *sec-bmelem-obj*). All parameters of the single *beam-recipe* object within the returned *beam-recipe* container are set from the corresponding parameters of the best *beam-recipe* object (symbol *best-bmrec-obj*). The *beam-name* parameter of the single *beam-recipe* object within the returned container is set to the symbol *sec-beam-name*. The *beam-azi* and *beam-slow* parameters of the single *beam-recipe* object within the returned *beam-recipe* container are reset by the inline common *libgobj* Scheme function `set-`



`gobj-attr` and the global Scheme function `nth-container` to the values held by symbols *az-d-o* and *sl-d-o*, respectively. Finally, this special *beam-recipe* object is referenced in the *amplitude-recipe* object within the *amplitude-recipe* container. This *amplitude-recipe* object is subsequently applied in the amplitude measurement process.

### Estimate Amplitude [12.7.3]

This subprocess measures the amplitudes. It exists within the innermost loop over sensors. It generates an estimate of the signal's amplitude from the bandpass-filtered waveforms for the current single sensor in accordance with the first *amplitude-recipe* object within the *amplitude-recipe* container. This estimation is performed by the *DFX/libamp* function `measure-amplitude`. This function is called by a Scheme language code such as:

```
(set! amp-obj (measure-amplitude wfdata-con (nth-container
0 amp-rec-con) ts-d-o te-d-o))
```

The function requires four arguments, which are described in Table 204.

**TABLE 204: FUNCTIONAL INTERFACE OF MEASURE-AMPLITUDE**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>(nth-container 0 amp-rec-con)</i>	input	GObj	<i>amplitude estimation processing recipe</i> object: amplitude estimation processing recipe to be applied
<i>ts-d-o</i>	input	real	start-time of window within which to perform waveform amplitude estimation
<i>te-d-o</i>	input	real	end-time of window within which to perform waveform amplitude estimation
n/a	returned value	GObj	<i>amplitude</i> object

▼ **Infrasonic Processing**

This function estimates the peak-to-trough amplitude, period, and time-of-estimate on a single cycle “lobe” of the bandpass-filtered waveforms for the current single sensor. In contrast with the FIR bandpass filtering applied to the waveforms used to establish detection, the bandpass filtering for amplitude estimation is performed in accordance with traditional IIR *libfilter* filtering algorithms (see [IDC5.2.1] for more information on filtering waveforms). The amplitude estimation is confined to the processing window defined by start-time symbol *ts-d-o* and end-time symbol *te-d-o*.

The estimated peak-to-trough amplitude is the maximum single-cycle value of peak-to-trough amplitude within the processing window. The returned object (symbol *amp-obj*) is an *amplitude* object. It is immediately transformed into a *dbamplitude* object by the local Scheme function *create-infra-detection-dbamplitude-from-amp*, and the returned *dbamplitude* object (symbol *dbamp-obj*) is put into a *dbamplitude* container (symbol *dbamp-con*).

**Select Best Measurement [12.7.4]**

This subprocess takes the single-sensor amplitude measurements for each sensor at the infrasonic station and returns the “best” measurement (the one to be reported). This subprocess exists between the innermost loop over sensors and the middle loop over *infra-detection* objects. It evaluates the *dbamplitude* container returned by the previous subprocess *Estimate Amplitude*. For the given current infrasonic detection, the evaluation starts by calculating the arithmetic mean of the amplitude measurements over those stations whose measurement times agree within the time-error of the original detection time. The single amplitude measurement that is closest to this mean amplitude is the amplitude measurement that is returned by the selection process. In fact, this single sensor's entire *dbamplitude* object is the *dbamplitude* object attributed to this infrasonic detection. This selection process is implemented by the local Scheme function *select-dbamplitude-object-to-report*. This function is called by a Scheme language code such as:

```
(set! out-dbamp-obj (select-dbamplitude-object-to-report
  dbamp-con time time-error))
```

The function requires three arguments, which are described in Table 205.

**TABLE 205: FUNCTIONAL INTERFACE OF SELECT-DBAMPLITUDE-OBJECT-TO-REPORT**

Argument Name	Usage	Type	Description
<i>dbamp-con</i>	input	container	<i>dbamplitude</i> objects
<i>time</i>	input	real	time of the peak of the energy packet underlying the current infrasonic detection
<i>time-error</i>	input	real	error estimate in the foregoing parameter (time)
n/a	returned value	GObj	<i>dbamplitude</i> object

This function returns a *dbamplitude* object (symbol *out-dbamp-con*), which corresponds to the **amplitude** table (see [IDC5.1.1Rev2]).

In support of rational human analyst review, the returned estimate of amplitude in this application is that single-sensor amplitude that is closest to the arithmetic mean of the amplitude estimates whose amplitude measurement times agree. The channel stored in the **amplitude** table record contains the station name of the single sensor whose amplitude estimate is reported.

### Detailed Description of Write Non-Redundant Detections [12.8]

*InfraDet* performs processing to eliminate both the redundant detections developed within a given single processing intervals as well as detections that are redundant relative to previous processing intervals. When the detections developed in a given processing interval for a given infrasonic station have had any redundant or already existing (in the database) members deleted, if any non-redundant detections remain, these are submitted to the appropriate tables of the database.

▼ **Infrasonic Processing**

The process *Write Non-Redundant Detections* accomplishes this. Figure 46 shows the architecture of the process. First, the process tests for an empty input container of db“entities”, which are to be written. It then writes the container of db“entities”, tests for unsuccessful writes, and rolls back the database if the write encountered any error.

The following subprocesses shown in Figure 46 are described in this section:

- *Test for Empty Input Container* [12.8.1]
- *Write Container* [12.8.2]
- *Test for Unsuccessful Write* [12.8.3]
- *Roll Back Database* [12.8.4]

**Test for Empty Input Container [12.8.1]**

This subprocess tests the input container of db“entities” (*dbarrival*, *dbdetection*, *dbinfra\_features*, and *dbamplitude*) to determine if the container is empty. It is implemented by the inline generic Scheme functions `if` and `null?`. If the container is empty, the subprocess calls the local Scheme function `depart-with-grace`. This function returns a formatted, informative warning message, rolls back the database via the global Scheme function `rollback-db` if there have already been successful writes to the database for this station and processing time interval, and then suspends further processing in this application and returns control to the calling program. If the container is not empty, then the subprocess *Write Container* is called.

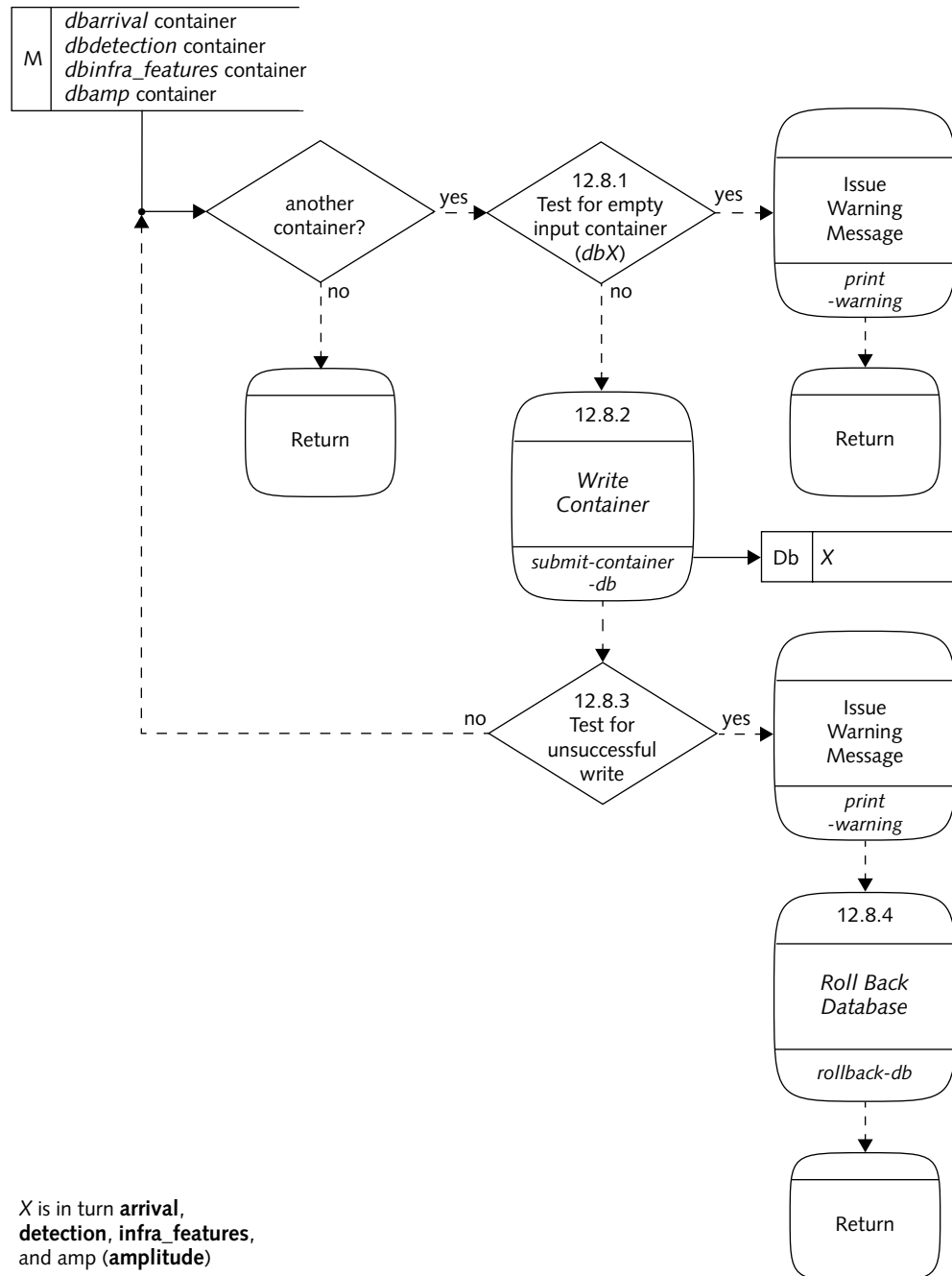


FIGURE 46. ARCHITECTURE OF WRITE NON-REDUNDANT DETECTIONS

▼ **Infrasonic Processing****Write Container [12.8.2]**

This subprocess writes the existing container of `db"entities"` to the database via the global Scheme function `submit-container-db`. This function is called by Scheme language code such as:

```
(set! returned-entity (submit-container-db db"entity"-con
    "entity"-table))
```

The function requires two arguments, which are described in Table 206.

**TABLE 206: FUNCTIONAL INTERFACE OF SUBMIT-CONTAINER-DB**

Argument Name	Usage	Type	Description
<code>db"entity"-con</code>	input	container	<code>db"entity"</code> objects
<code>"entity"-table</code>	input	character-string	name of database table in which "entity" records are stored
n/a	returned value	logical scalar	indicates success ( <code>t</code> ) or failure ( <code>nil</code> )

This function assumes that the database is opened, and when finished it does not commit the database. It causes each object within the input container of `db"entity"` objects (symbol `db"entity"-con`) to be written as a record to the database table whose name is indicated by the argument with symbol `"entity"-table`. There is a one-to-one correspondence between parameters in the `db"entity"` object and attributes (columns) in the database record. If the submission to the database is successful, `submit-container-db` returns `t`; otherwise, it returns `nil`.

**Test for Unsuccessful Write [12.8.3]**

This subprocess tests the returned value from the global Scheme function `submit-container-db` via the inline intrinsic Scheme functions `if` and `null?` to determine whether or not the write was successful. If the write was not successful, the local Scheme function `depart-with-grace` is called. This function returns a

formatted, informative warning message, rolls back the database via the global Scheme function `rollback-db` if there have already been successful writes to the database for this site and processing time interval, suspends further processing in this application, and returns control to the calling program. If the write was successful, then the subprocess *Test for Empty Input Container* is called for the next container of `db` "entity" objects to be written to the database. When the *dbarrival*, *dbdetection*, and *dbinfra\_features* have all been written successfully, the database is "committed" by the global Scheme function `commit-db`. Thereafter, the subprocess *Test for Empty Input Container* is called for the last container (the *dbamplitude* container) to be written to the database. When the *dbamplitude* has been written successfully, the database is again committed by the global Scheme function `commit-db`.

#### Roll Back Database [12.8.4]

This subprocess returns a formatted warning message and rolls back the database to the state it had just after the most recent `commit-db`. It is implemented by the local Scheme function `depart-with-grace`. This function is called by a Scheme language code such as:

```
(set! returned-entity (depart-with-grace message-string
roll))
```

The function requires two arguments, which are described in Table 207.

**TABLE 207: FUNCTIONAL INTERFACE OF DEPART-WITH-GRACE**

Argument Name	Usage	Type	Description
<i>message-string</i>	input	character-string	text of formatted message to be returned
<i>roll</i>	input	integer	zero if rollback is not to be performed, unity if rollback is to be performed
n/a	returned value	logical scalar	indicates success ( <code>t</code> ) or failure ( <code>nil</code> )

▼ **Infrasonic Processing**

This function returns the warning message text referenced by the symbol *message-string*. If the argument *roll* is unity in value, the database is rolled back by the global Scheme function `rollback-db`, further processing in this application is terminated, and control is returned to the calling program; otherwise (*roll* equal to zero in value), the database is left in its current state, further processing in this application is terminated, and control is returned to the calling program.



## Chapter 7: Administrative Processing

This chapter describes the detailed design of *DFX* administrative processing and includes the following topics:

- Segment Archiving [13]
- Quality Control Statistics [14]

## Chapter 7: Administrative Processing

### SEGMENT ARCHIVING [13]

*Segment Archiving (SegArch)* is a DFX Scheme application that saves origin beams for all arrivals associated with each of the origins that occur within a given time interval. The source code resides in the file `DFX-segarch.scm`. *SegArch* is typically called in a post-analysis processing pipeline. It is run by specifying the network, start-time and end-time, and the `DFX-segarch.par` file on the command line. The results of *SegArch* are written to the **wfdisc** table and, if desired, the **wftag** table.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Perform Segment Archiving* [13.3]

## General Description

*SegArch* is the *DFX* application that automatically generates and saves origin beams from seismic arrays based upon the origins in the processing time interval. Because the waveform time segments and steering information used to form the beams is derived from event origins, these origin beams can be relied upon to contain the signal arrival if it is sufficiently energetic.

## Architecture

*SegArch* consists of three principal processes (see Figure 47). The application is implemented by a single pass through two initialization processes followed by a nested pair of loops in which the third process performs segment archiving. The first initialization process queries for origins (in the time interval) by filling a container with *dboorigin* objects. The process *Query For Origins (In Interval)* is implemented by `query-for-dboorigin-container`, which, if successful, fills a container with *dboorigin* objects for all of the seismic origins available for the processing time interval. For general information on how database queries are implemented, see “Querying the Database” on page 41. The second initialization process initializes sites by acquiring all of the site information. It is implemented by the local Scheme function `initialize-net-sites`, which follows the general site initialization model described in “Initializing Sites” on page 46. The outer loop runs over the origins, the inner loop runs over the stations associated with the given network, and the process *Perform Segment Archiving* generates and stores the origin beams. This process is implemented by the local Scheme function `process-origins`.

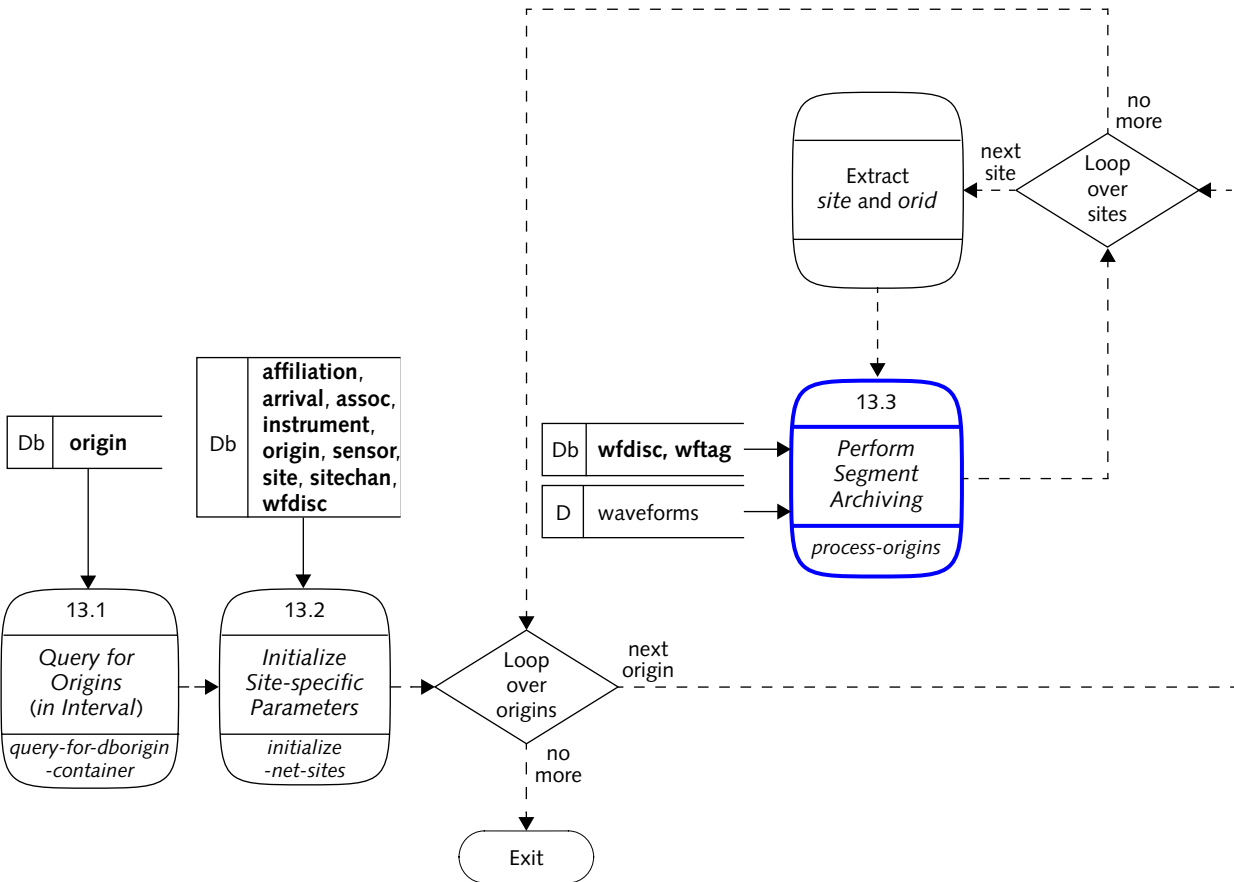


FIGURE 47. ARCHITECTURE OF SEGARCH

## Input/Output

The input CVAR datastore parameters used in the Scheme code are listed in Table 208.

**TABLE 208: INPUT PARAMETERS FOR SEGMENT ARCHIVING PROCESSING**

Par Name	Source	Default Value	Description
<i>net</i>	command line	<none>	network of station to process
<i>start-time</i>	command line	<none>	start-time of the processing interval
<i>end-time</i>	command line	<none>	end-time of the processing interval
<i>delta1</i>	par file	<none>	distance (degrees) separating regional from teleseismic
<i>delta2</i>	par file	<none>	intermediate distance (degrees)
<i>data-offset-len</i>	par file	50.0	duration of data to add to the ends of the waveform interval to allow for filter start-up and taper-off
<i>wfdisc-extension-len</i>	par file	86400.0	maximum time duration of ".w" files in which binary format waveform time series are stored on the UNIX filesystem
<i>out-database-account</i>	par file	<none>	database account for the output
<i>data-output-wftag</i>	par file	0	if non-zero, wftag of the output data
<i>affiliation-table</i>	par file	affiliation	table containing station and network affiliations
<i>in-arrival-table</i>	par file	arrival	input table containing signal arrival information
<i>in-assoc-table</i>	par file	assoc	input table containing association information

## ▼ Administrative Processing

**TABLE 208: INPUT PARAMETERS FOR SEGMENT ARCHIVING  
PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>instrument-table</i>	par file	instrument	table containing generic (default) calibration information about a station
<i>origin-table</i>	par file	origin	table containing origin information
<i>sensor-table</i>	par file	sensor	table containing specific calibration information for physical channels
<i>site-table</i>	par file	site	table containing site-location information
<i>sitechan-table</i>	par file	sitechan	table containing station-channel information
<i>in-wfdisc-table</i>	par file	wfdisc	input table containing waveform file header and descriptive information
<i>out-wfdisc-table</i>	par file	wfdisc	output table containing waveform file header and descriptive information generated by this application
<i>out-wftag-table</i>	par file	wftag	output table containing waveform mapping information generated by this application
<i>dfx-site-event-file</i>	par file	<none>	name of the file containing the site-specific recipes for generating the origin beams
<i>origin-output-beam-tirec-list</i>	par file	<none>	time-interval recipes for output beams
<i>origin-observed-amprec-list</i>	par file	<none>	amplitude recipes for observed arrivals/phases for given origins
<i>origin-theoretical-amprec-list</i>	par file	<none>	amplitude recipes for theoretical arrivals

**TABLE 208: INPUT PARAMETERS FOR SEGMENT ARCHIVING PROCESSING (CONTINUED)**

Par Name	Source	Default Value	Description
<i>origin-output-beamrec-list</i>	par file	<none>	beam recipes for output beams
<i>beamrec-list1</i>	par file	<none>	list of beam recipes for regional distance
<i>beamrec-list2-3</i>	par file	<none>	list of beam recipes for teleseismic distances

All results from this application are written to the database tables described in Table 209. There are database records as well as files of the time-series samples of the origin beams, referred to as “.w” files. These “.w” files are written to the UNIX filesystem in the directory and filenames delineated by the *dir* and *dfile* attributes of the **wfdisc** table. The directory name is of the form */base/YYYY/JJJ/* where *JJJ* is the three-digit Julian day of the year (with lead zeros as required) and *YYYY* is the four-digit year for the given input waveforms used to generate the origin beam. The base directory path is set in the **DFX-segarch.par** file by the *data-output-base-directory* parameter. This application returns an exit status code whose value indicates whether or not the processing was successful.

Table 242 on page 452 indicates the specific attributes written for these database tables.

**TABLE 209: DATA PRODUCED BY SEGMENT ARCHIVING PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>wfdisc</b>	database table	origin beams generated
<b>wftag</b>	database table	maps origins to beams
“.w”	binary files	origin-beam time-series samples

## ▼ Administrative Processing

**Processes**

The following processes shown in Figure 47 on page 408 are described in this section:

- *Query for Origins (in Interval)* [13.1]
- *Initialize Site-specific Parameters* [13.2]
- *Perform Segment Archiving* [13.3]

**Query for Origins (in Interval) [13.1]**

The process *Query For Origins (In Interval)* constructs a suitable database query, which is submitted to the **origin** table, and returns the results as a *dborigin* container. It is implemented by a call to the local Scheme function `query-for-dborigin-container`. (For general information on how database queries are implemented see “Querying the Database” on page 41.) This function is called by the Scheme language code:

```
(query-for-dborigin-container)
```

The function requires no arguments. It acquires the start-time and end-time, and the **origin** table from the CVAR datastore. It then constructs the database query, submits the query to the database via the global Scheme function `create-and-fill-container`, and returns a *dborigin* container.

**Initialize Site-specific Parameters [13.2]**

Process *Initialize Sites* acquires all of the site information for the stations belonging to the network specified by the parameter *net*. It is implemented by a call to the local function `initialize-net-sites`. The function follows the general site initialization model described in “Initializing Sites” on page 46. It is called by the Scheme language code:

```
(initialize-net-sites dborigin-container)
```

The function requires one argument, which is described in Table 68 on page 155.



This function creates an *initsite* container in which to return the *initsite* objects. It then acquires local variables by accessing the CVAR datastore and fills the *initsite* container with the *initsite* objects consisting of *dbsite* and *recipe* objects. The local Scheme function `query-for-network-site-container` obtains the *dbsite* objects for the specified network. The objects are used by the local Scheme function `initialize-net-site` to create the *initsite* objects. The *recipe* objects, which are attached to the *initsite* fields, are obtained from the CVAR datastore.

After the *initsite* container is populated, the function `initialize-net-sites` initializes the travel-time tables for all seismic phases from all sites via the local Scheme function `initialize-net-tt-tables`. The function reads all relevant **arrival**, **association**, **origin**, **affiliation**, and **site** information from the corresponding database tables into a *dbarsoc* container via the local Scheme function `query-for-dbarsoc-container`. The function then computes time intervals for all sites for all of the origins in the *dborigin* container via the local Scheme function `compute-net-ti`. The function `initialize-net-sites` then acquires the *wfdata* and corresponding *wfmems* affiliated with the network sites via the local Scheme functions `query-for-wfdata-container` and `query-for-wfmem-container`, which use the **site**, **sensor**, **instrument**, **sitechan**, **affiliation** and **wfdisc** database tables.

Finally, for each site in the *initsite* container, `initialize-net-sites` extracts initialization information and sets the *arsoc* container, *wfdata* container, and *wfmem* containers within the *initsite* object for that specific site. It returns an *initsite* container. The contents of an *initsite* object specific to the *SegArch* application are provided in the 13th column (*SegArch*) in Table 8 on page 50 (see “Contents of Initsite Objects” on page 46).

### Perform Segment Archiving [13.3]

The process *Perform Segment Archiving* generates and saves the origin beams required for segment archiving. It writes them to the database and the UNIX file-system. This process is implemented by the local Scheme function `process-origin`, which is called from the local Scheme function `process-origins` (note the

## ▼ Administrative Processing

distinction in function names). The latter function simply provides the loop over origins (outer loop) and passes arguments. It is called by the Scheme language code:

```
(process-origins initsite-container origin-container)
```

This function requires two arguments, which are described in Table 210.

**TABLE 210: FUNCTIONAL INTERFACE OF PROCESS-ORIGINS**

Argument Name	Usage	Type	Description
<i>initsite-container</i>	input	container	<i>initsite</i> objects
<i>origin-container</i>	input	container	<i>dborigin</i> objects
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The processing is performed by the local Scheme function `process-origin`, which operates on each *dborigin* object. Function `process-origin` is called by the Scheme code:

```
(process-origin initsite-container origin)
```

This function requires two arguments, which are described in Table 211.

**TABLE 211: FUNCTIONAL INTERFACE OF PROCESS-ORIGIN**

Argument	Usage	Type	Description
<i>initsite-container</i>	input	container	<i>initsite</i> objects
<i>origin</i>	input	GObj	<i>dborigin</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

For each origin this function performs a loop over the sites referenced in the *initsite* container. It performs the following subprocesses for each origin and site:

1. Extract parameters.

2. Calculate the distance-based time interval.
3. Calculate station-to-event azimuth and slowness.
4. Generate distance-based beam recipes.
5. Acquire waveforms.
6. Check waveform quality.
7. Generate beams (and write their time-series data samples to the UNIX filesystem).
8. Write beams to the database (**wfdisc** and **wftag**).

The next section provides more information about this function.

### Detailed Description of Perform Segment Archiving [13.3]

This process is accomplished by a loop over origins in which the inner-most function, `process-origin`, accomplishes segment archiving for each given origin and site.

Figure 48 shows the architecture of this process. The function `process-origin` implements a loop over sites, which encloses a single pass through eight subprocesses.

The following subprocesses shown in Figure 48 are described in this section:

- *Extract Parameters [13.3.1]*
- *Calculate Distance-based Time Intervals [13.3.2]*
- *Calculate Station-to-Event Distance [13.3.3]*
- *Generate Distance-based Beam Recipes [13.3.4]*
- *Acquire Waveforms [13.3.5]*
- *Check Waveform Quality [13.3.6]*
- *Generate Beams [13.3.7]*
- *Write Beams to Database [13.3.8]*

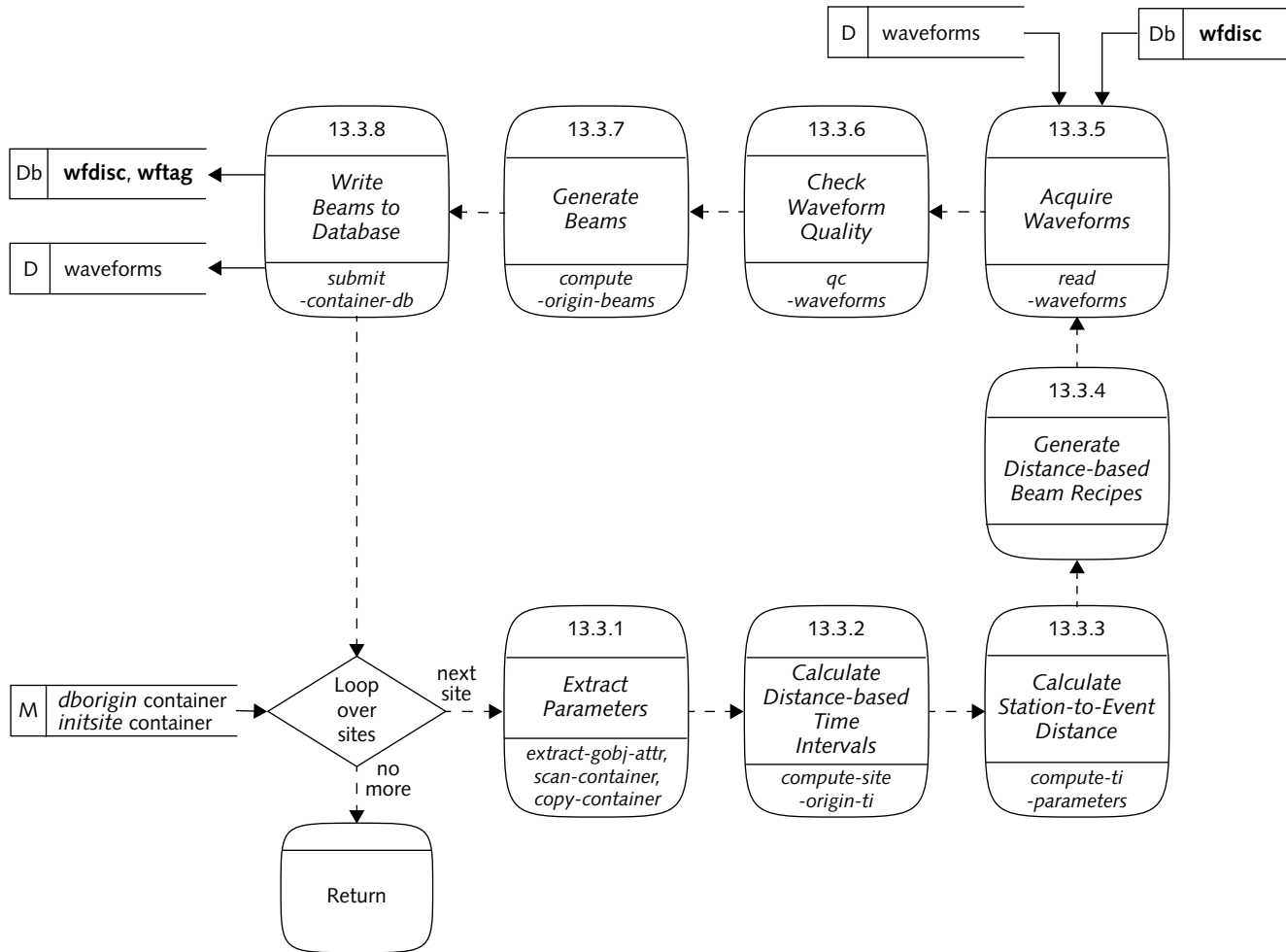


FIGURE 48. ARCHITECTURE OF PERFORM SEGMENT ARCHIVING

**Extract Parameters [13.3.1]**

For each processing time interval and origin, a variety of parameter values are needed to perform segment archiving. The subprocess *Extract Parameters* obtains these values, as well as other required symbols, objects, and containers of objects from the input *initsite* container via the inline Scheme intrinsic function `define`, the common *libgobj* function `extract-gobj-attr`, and global functions `scan-container` and `copy-container`.

**Calculate Distance-based Time Intervals [13.3.2]**

A time interval for acquiring waveforms is estimated based on the event-to-station distance and the phase specified in the time-interval recipe. The subprocess *Calculate Distance-based Time Intervals* applies the *DFX/libutil* function `compute-site-origin-ti` and the inline intrinsic Scheme functions `define`, `car`, and `cadr` to determine the requisite time interval. The *DFX/libutil* function `compute-site-origin-ti` is called by the Scheme language code:

```
(set! ti-list (compute-site-origin-ti dbsite dborigin
  origin-tirec-con))
```

The function requires three arguments, which are described in Table 212. This table is quite similar to Table 14 on page 64; however, the second argument is a *dborigin* object and the third argument is a *origin time-interval recipe* container.

**TABLE 212: FUNCTIONAL INTERFACE OF COMPUTE-SITE-ORIGIN-TI**

Argument Name	Usage	Type	Description
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>origin-tirec-con</i>	input	container	<i>time-interval processing recipe</i> objects (for required origin beams)
n/a	returned value	list	reals: start-times and end-times for intervals

## ▼ Administrative Processing

This function returns a list of time intervals for each recipe in the *origin-beam time-interval processing recipe* container. The least and greatest start-times and end-times, respectively, are extracted from the returned list and are used to define the time interval applied in acquiring the waveforms for all sites and for all origins. This time interval is then extended on each end (that is, both an earlier beginning and a later ending) by an amount specified in the parameter *data-offset-len*. This increase ensures that edge effects do not degrade the beam that is to be formed.

### Calculate Station-to-Event Distance [13.3.3]

This subprocess calculates the station-to-event distance via the *DFX/libutil* function *compute-ti-parameters*. This distance is used later to determine which type of beam recipe to use in forming the origin beam. The function *compute-ti-parameters* is called by the Scheme language code:

```
(set! se-list (compute-ti-parameters dbsite dborigin
  origin-tirec-con "P"))
```

The function requires four arguments, which are described in Table 213.

**TABLE 213: FUNCTIONAL INTERFACE OF COMPUTE-TI-PARAMETERS**

Argument Name	Usage	Type	Description
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object
<i>origin-tirec-con</i>	input	container	<i>time-interval processing recipe</i> objects (for required origin beams)
"P"	input	character-string	phase of first arrival
n/a	returned value	list	reals and character-string: start-times and end- times, station-to-event distance, azimuths, magnitude-slowness, and model information

This function returns a list consisting of a character-string identifying the travel-time model used and several real-valued parameters, including the station-to-event distance (in degrees).

### Generate Distance-based Beam Recipes [13.3.4]

This subprocess extracts the distance-specific beam recipes from the *beam-recipe* container into a new container (symbol *new-beamrec-con*). The distance-specific beam names are obtained from the user parameters *beamrec-list1* and *beamrec-list2-3*.

### Acquire Waveforms [13.3.5]

This subprocess acquires waveforms for the appropriate processing interval given the arrival phase and station-to-event distance. It is implemented by the *DFX/lib-data* function *read-waveforms*. This function is called by the Scheme language code:

```
(set! wfdata-con (read-waveforms wfmem-con wfdata-con t1
                                t2))
```

The function requires four arguments, which are described in Table 214.

**TABLE 214: FUNCTIONAL INTERFACE OF READ-WAVEFORMS**

Argument	Usage	Type	Description
<i>wfdata-con</i>	input and output	container	<i>wfdata</i> objects
<i>wfmem-con</i>	input and output	container	<i>wfmem</i> objects
<i>t1</i>	input	real	start-time of this processing interval
<i>t2</i>	input	real	end-time of this processing interval
n/a	returned value	container	<i>wfdata</i> objects

## ▼ Administrative Processing

This function produces its result in two fashions. First, as a side-effect, the input arguments *wfmem-con* and *wfdata-con* are altered in accordance with the acquisition of the waveforms. Second, the function `read-waveforms` returns the altered *wfdata* container (symbol *wfdata-con*), which is null if an error occurred.

This subprocess acquires waveforms by the standard method in *DFX* (see “Reading Waveforms” on page 58).

### Check Waveform Quality [13.3.6]

This subprocess checks the quality of and repairs minor defects in the acquired waveforms via the *DFX/libqc* function `qc-waveforms`. The subprocess uses the standard method in *DFX* for checking the quality of waveforms (see “Checking Quality of Waveforms” on page 69).

### Generate Beams [13.3.7]

This subprocess generates the origin beams for the current processing interval, origin, and site via the local Scheme function `compute-origin-beams`. This function is called by the Scheme language code:

```
(set! origin-beam-list (compute-origin-beams wfdata-con
      dbsite dborigin new-beamrec-con tirec-con))
```

The function requires five arguments, which are described in Table 215.

**TABLE 215: FUNCTIONAL INTERFACE OF COMPUTE-ORIGIN-BEAMS**

Argument Name	Usage	Type	Description
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects
<i>dbsite</i>	input	GObj	<i>dbsite</i> object
<i>dborigin</i>	input	GObj	<i>dborigin</i> object



TABLE 215: FUNCTIONAL INTERFACE OF COMPUTE-ORIGIN-BEAMS

Argument Name	Usage	Type	Description
<i>new-beamrec-con</i>	input	container	<i>beam-recipe</i> objects
<i>tirec-con</i>	input	container	<i>time-interval recipe</i> objects
n/a	returned value	list	<i>dbwfdisc</i> and <i>dbwftag</i> containers

The function `compute-origin-beams` performs the following steps:

1. Determine the number of *beam-recipe* objects that exist, and acquire this number of unique *wfids* from the `lastid` database table.
2. Fill the *dbwfdisc* container with the *beam* objects returned by the `DFX/libio` function `create-db-origin-beam`.
3. Return a list consisting of the *wfdisc* and *wftag* containers. The latter is empty if the parameter *data-output-wftag* is not set to unity.

The function `create-db-origin-beam` performs the following steps:

1. Compute beams based on the vector-slowness predicted by the station and origin locations and the given arrival phase for each beam recipe in the input *beam-recipe* container.
2. Write the beam time-series data samples to the appropriate directory and filename of the UNIX filesystem.
3. Create and fill the *dbwfdisc* output container.
4. If required, create a *dbwftag* container corresponding to the *dbwfdisc* container.
5. Return a list consisting of the *dbwfdisc* and *dbwftag* containers.

The symbols *dbsite* (site information), *dborigin* (origin information), and *tirec-con* (time-interval recipe information) are used to compute the time intervals over which the origin beams are computed.

## ▼ Administrative Processing

**Write Beams to Database [13.3.8]**

This subprocess writes the **wfdiscs** and the **wftags** to the database. The inline intrinsic Scheme functions `car` and `cadr` are used to separate the *dbwfdisc* and *dbwftag* containers from the list returned by the local Scheme function `compute-origin-beams`. The global Scheme function `submit-container-db` is used to save the *dbwfdisc* and *dbwftag* containers in the appropriate database tables. The specific table names are provided by the CVAR datastore parameters *out-wfdisc-table* and *out-wftag-table* by the inline global Scheme function `mstspar`. For more information about submitting results to the database see “Writing to the Database” on page 44.

## QUALITY CONTROL STATISTICS [14]

*Quality Control Statistics (QCStats)* is a *DFX* Scheme application that produces statistical measures of the attributes determined while checking the quality of raw waveforms at a given site. The source code resides in the file `DFX-qcstats.scm`. In contrast to the other 13 *DFX* applications, *QCStats* is invoked only from within the three automatic waveform detection applications: *SeisDet* (seismic), *HydroDet* (hydroacoustic), and *InfraDet* (infrasonic). The results of *QCStats* are written to the **qcstats** table of the database.

The application's top-level architecture is defined and illustrated in the following sections. Each process in the top-level design is described in general terms. Processes that require further explanation have detailed descriptions that follow the general sections. These processes are indicated by the heavy lines in the architectural diagram. The detailed descriptions decompose the process into its constituent subprocesses. This description includes the following topics:

- General Description
- Architecture
- Input/Output
- Processes
- Detailed Description of *Identify Missing Waveforms* [14.1]
- Detailed Description of *Calculate Waveforms Availability* [14.3]

### General Description

This application evaluates the attributes of a set of waveforms that have been subject to standard quality-control processing. It computes statistical measures concerning the quality of the waveforms and writes them to the **qcstats** database table.

### Architecture

*QCStats* is implemented by two principal processes (see Figure 49). The first process identifies missing waveforms by determining which station-channel pairs are not present in the overall set of waveforms. Information about these missing data

## ▼ Administrative Processing

station-channel pairs is written to the **qcstats** database table. The second process occurs in a loop over those station-channel pairs for which waveforms are available in the set of quality-checked waveforms. For each station-channel pair, statistical measures are computed and the discrepancy between the requested and available data is calculated. Results are written to the **qcstats** database table.

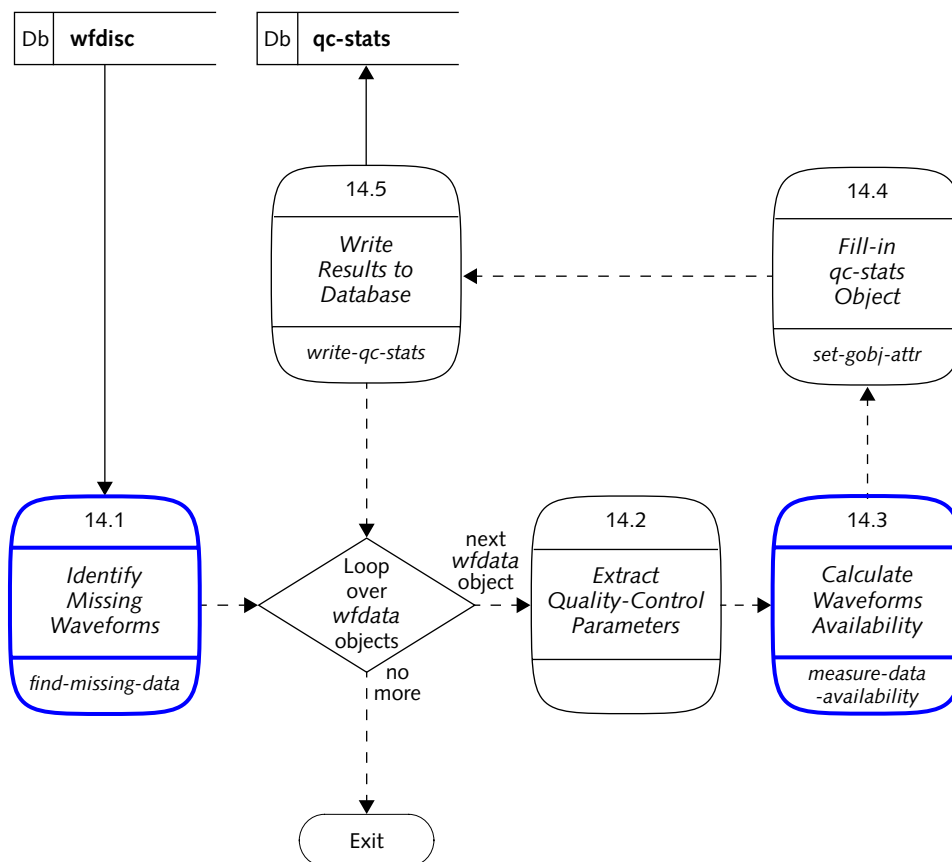


FIGURE 49. ARCHITECTURE OF QCSTATS

### Input/Output

Table 216 lists the input CVAR datastore parameters used in the Scheme code.

**TABLE 216: INPUT PARAMETERS FOR QUALITY CONTROL STATISTICS PROCESSING**

Par Name	Source	Default Value	Description
<i>processing-interval-list</i>	argument #1 of application	<none>	list of start-times and end-times for these waveforms earlier and later than requested times, respectively, to account for extra waveforms required to avoid signal processing edge effects
<i>sta-chan-pairs</i>	argument #2 of application	<none>	list of station and channel codes at this site
<i>wfdata-con</i>	argument #3 of application	<none>	<i>wfdata</i> objects container
<i>sta</i>	par file	<none>	code of station to process
<i>start-time</i>	par file	<none>	requested start-time of waveforms
<i>end-time</i>	par file	<none>	requested end-time of waveforms
<i>qcstats-cull-sta-chan-list</i>	par file	<none>	list of station-channel pairs whose waveforms are dropped due to excessive deficiency
<i>lastid-table</i>	par file	<i>lastid</i>	name of database table that keeps track of unique identification numbers for each category of data entry
<i>out-qcstats-table</i>	par file	<none>	name of database table that receives output results from this application

## ▼ Administrative Processing

All results from this application are written to the database table described in Table 217.

**TABLE 217: DATA PRODUCED BY QUALITY CONTROL STATISTICS PROCESSING**

Name	Category	Description
returned value	application exit status	indicates success or failure
<b>qcstats</b>	database table	holds statistical measures about the waveforms for this station and processing time interval

Table 243 on page 452 indicates the specific attributes written to this database table. This application returns a status code indicating whether or not the processing was successful.

Because the function `process-qcstats` is invoked only from within other *DFX* applications, rather than from a process control pipeline, the nature of the invocation is identical to the Scheme language code for invoking any other Scheme language function, as in:

```
(process-qcstats processing-interval-list sta-chan-pairs
  wfdata-con)
```

The function requires three arguments, which are listed in Table 218.

**TABLE 218: FUNCTIONAL INTERFACE OF PROCESS-QCSTATS**

Argument Name	Usage	Type	Description
<i>processing-interval-list</i>	input	list	reals consisting of the start-time and end-time of these waveforms
<i>sta-chan-pairs</i>	input	list	pairs of character-strings: codes of station-channel pairs at this station
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects

The intended output from this function is written directly to the database (see Table 217 on page 426). This function acts as a subroutine and has no meaningful return value.

The symbol *processing-interval-list* provides the start-time and end-time values of the processing interval expressed as a two-element list. The symbol *sta-chan-pairs* provides a list of pairs of character-strings (one pair for each combination of station code and channel code for the station-channel pairs to be processed). For example, this list may resemble ((LS01 sd) (LS02 sd) (LS04 sd)) where three station-channel pairs all have channel character-strings "sd" and three station character-strings as "LS01", "LS02", and "LS04". The symbol *wfdata-con* is a *wfdata* container for each station-channel pair. Each *wfdata* object includes a quality-control mask, and a *wfmem* object itself points to a data vector (the vector of waveforms time-series samples). The mask is a structure that is associated with the data vector and contains quality-control information concerning the waveforms for the given station-channel pair.

## Processes

The following processes shown in Figure 49 on page 424 are described in this section:

- *Identify Missing Waveforms [14.1]*
- *Extract Quality-control Parameters [14.2]*
- *Calculate Waveforms Availability [14.3]*
- *Fill-in qc-stats Object [14.4]*
- *Write Results to Database [14.5]*

### Identify Missing Waveforms [14.1]

This process identifies the station-channel pairs for which waveforms do not exist. This process is implemented by a call to the local Scheme function `find-missing-data`. It compares the station-channel pairs with those it finds in the waveform container. An entry to the `qcstats` table is made for the missing station-channel pairs. This function is called by the Scheme language code:

## ▼ Administrative Processing

```
(find-missing-data qc-stats-obj sta-chan-pairs
  wfdata-con)
```

The function requires three arguments, which are described in Table 219.

**TABLE 219: FUNCTIONAL INTERFACE OF FIND-MISSING-DATA**

Argument Name	Usage	Type	Description
<i>qc-stats-obj</i>	output	GObj	<i>qc-stats</i> object
<i>sta-chan-pairs</i>	input	list	pairs of character-strings of the codes of the paired stations and channels at this site
<i>wfdata-con</i>	input	container	<i>wfdata</i> objects for waveforms that have been checked for quality

This function's results are returned as a side-effect by modifying the contents of the first argument, namely the *qc-stats* object denoted by the symbol *qc-stats-obj*. It uses the character-strings for the station and channel codes of a given station-channel pair and checks existence in the container *wfdata-con*. Missing channel information is written to the database. If there are no missing station-channel pairs, the *qc-stats* object is NULL on return.

For more information about this function see "Detailed Description of Identify Missing Waveforms [14.1]" on page 432.

### **Extract Quality-control Parameters [14.2]**

This process extracts the quality-control attributes from the mask (for each object in the *wfdata* container) so that these attributes are available to be written to the *qc-stats* object (see "Fill-in qc-stats Object [14.4]" on page 430). This process is implemented by inline Scheme code within the function `process-qcstats`. The



common *libgobj* Scheme function `extract-gobj-attr` performs the required processing. Information is extracted from the *wfdata* object and its mask. Table 220 lists the information extracted from each mask.

**TABLE 220: INFORMATION EXTRACTED FROM EACH WFDATA OBJECT'S MASK**

Parameter Name	Type	Description
<i>nseg</i>	integer	number of masked waveform segments
<i>maskedtime</i>	real	total time-duration of waveforms with masked segments
<i>pointspike</i>	real	total time-duration of waveforms masked due to single-sample spikes
<i>spike</i>	real	total time-duration of waveforms masked due to multi-sample spikes
<i>nconstseg</i>	integer	total number of waveform segments with constant value
<i>consttime</i>	real	total time-duration of waveform segments with constant value
<i>avgconst</i>	real	average time-duration of waveform segments with constant value
<i>stdconst</i>	real	standard deviation of time-duration of waveform segments with constant value
<i>dropped</i>	integer	valued 1 if the waveform object for this station-channel pair is to be removed from the set of quality-control evaluated waveforms valued 0, otherwise

### Calculate Waveforms Availability [14.3]

This process compares the available time-duration of the quality-checked waveforms with the requested time-duration of the processing interval (provided by access to the CVAR datastore). It sets the “missing” attribute of the *qc-stats* object

## ▼ Administrative Processing

to the difference in these time-durations (*requested-available*). This process is implemented by a call to the local Scheme function `measure-data-availability`. The function is called by the Scheme language code:

```
(measure-data-availability qc-stats-obj wfdata-obj)
```

The function requires two arguments, which are described in Table 221.

**TABLE 221: FUNCTIONAL INTERFACE OF MEASURE-DATA-AVAILABILITY**

Argument Name	Usage	Type	Description
<i>qc-stats-obj</i>	output	GObj	<i>qc-stats</i> object
<i>wfdata-obj</i>	input	GObj	<i>wfdata</i> object

The results of this function are returned as a side-effect; symbol *qc-stats-obj* has its missing attribute set to the difference between the requested waveforms time-duration and the amount of waveforms in the *wfdata* object.

For more information about this function see “Detailed Description of Calculate Waveforms Availability [14.3]” on page 437.

#### Fill-in *qc-stats* Object [14.4]

This process fills in the attributes of the *qc-stats* object, which is produced for each *wfdata* object in the quality-checked *wfdata* container. It is implemented by inline Scheme code using the common *libgobj* Scheme function `set-gobj-attr` in the loop over waveforms objects within the function `process-qcstats`. Table 222 lists the nature of the information stored in the *qc-stats* object for each station-channel pair.

**TABLE 222: INFORMATION STORED IN EACH QC-STATS OBJECT**

Parameter	Type	Description
<i>detime</i>	real	start-time of these waveforms
<i>detendtime</i>	real	end-time of these waveforms
<i>sta</i>	character-string	station code
<i>chan</i>	character-string	channel code
<i>dropped</i>	integer	valued 1, if the <i>wfdata</i> object is to be removed from the <i>quality-checked waveform</i> container valued 0, otherwise
<i>nseg</i>	integer	number of waveform segments masked
<i>masked</i>	real	total time-duration of masked segments
<i>pointspike</i>	real	total time-duration of single-sample spikes
<i>spike</i>	real	total time-duration of multi-sample spikes
<i>nconstseg</i>	integer	number of waveform segments with constant value
<i>const</i>	real	total time-duration of waveform segments with constant value
<i>avgconstval</i>	real	average time-duration of waveform segments with constant value
<i>stdconstval</i>	real	standard deviation of time-duration of waveform segments with constant value

**Write Results to Database [14.5]**

This process writes a *qc-stats* object to the database. The database write is skipped if the station-channel pair is a member of the parameter *qcstats-cull-sta-chan-list*. Otherwise, attributes of the *qc-stats* object are set, and the *qcstatid* value is set. Then the *qc-stats* object is written to the **qcstats** database table. This process is implemented by a call to the local Scheme function `write-qcstats`, which uses

## ▼ Administrative Processing

the global Scheme function `submit-container-db` to write to the **qcstats** database table. For more information on submitting results to the database see “Writing to the Database” on page 44. This function is called by the Scheme language code:

```
(write-qcstats qc-stats-obj)
```

The function requires one argument, which is described in Table 223.

**TABLE 223: FUNCTIONAL INTERFACE OF WRITE-QCSTATS**

Argument Name	Usage	Type	Description
<i>qc-stats-obj</i>	input	GObj	<i>qc-stats</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The *qc-stats* object provides all of the information written to the **qcstats** database table.

### Detailed Description of Identify Missing Waveforms [14.1]

This process examines the list of requested station-channel pairs and compares each of these with the station-channel pairs found in the *quality-checked waveforms* container in search of an exact match. Where none exists, a record concerning that requested station-channel pair is written to the **qcstats** database table indicating that waveforms are missing. This process is implemented by a call to the local Scheme function `find-missing-data`.

Figure 50 shows the architecture of this function. This function is implemented by five principal subprocesses. First, values are acquired for symbols required in performing this process. The remaining four subprocesses are contained in a loop over the full set of requested station-channel pairs. For each station-channel pair, the *quality-checked waveforms* container is searched to determine whether or not an

exactly matching station-channel pair exists therein. If not, then appropriate attributes of the *qc-stats* object for this station-channel pair are filled-in. Finally, this *qc-stats* object is written to the **qcstats** database table.

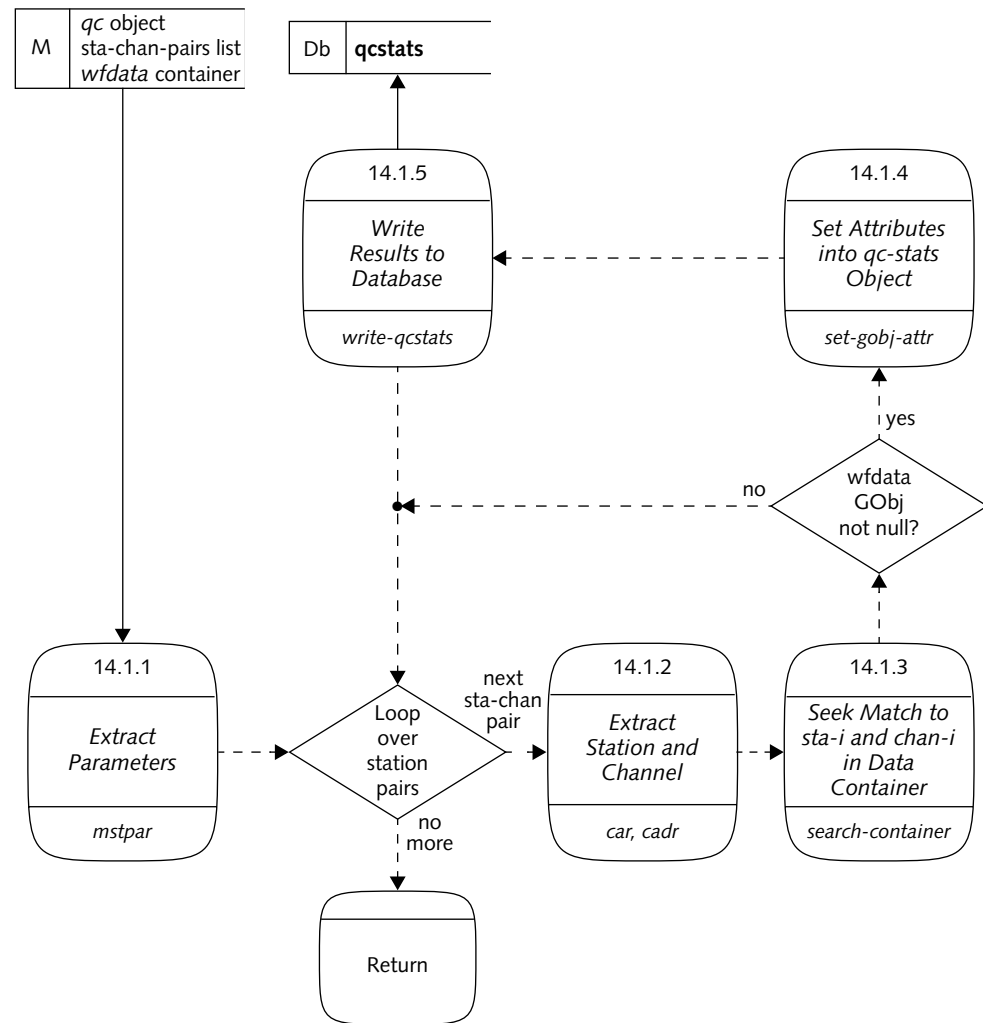


FIGURE 50. ARCHITECTURE OF IDENTIFY MISSING WAVEFORMS

## ▼ Administrative Processing

The following subprocesses shown in Figure 50 are described in this section:

- *Extract Parameters [14.1.1]*
- *Extract Station and Channel [14.1.2]*
- *Seek Match to sta-i and chan-i in Data Container [14.1.3]*
- *Set Attributes into qc-stats Object [14.1.4]*
- *Write Results to Database [14.1.5]*

### Extract Parameters [14.1.1]

This subprocess acquires values for the requested start-times and end-times required to calculate the quantity of missing waveforms. The inline global Scheme function `mstpar` is used to extract values for these parameters from the CVAR datastore. The parameters, symbols *t1* and *t2*, are listed in Table 224.

**TABLE 224: INTERNAL PARAMETERS OF FIND-MISSING-DATA**

Symbol	Type	Description
<i>t1</i>	real	requested start-time of these waveforms
<i>t2</i>	real	requested end-time of these waveforms
<i>sta-i</i>	character-string	station code of this station-channel pair
<i>chan-i</i>	character-string	channel code of this station-channel pair

### Extract Station and Channel [14.1.2]

This subprocess acquires the station and channel codes required for each station-channel pair in the list of requested station-channel pairs. The inline intrinsic Scheme functions `car` and `cadr` are used to extract values for these parameters. The parameters, symbols *sta-i* and *chan-i*, are listed in Table 224.

### Seek Match to *sta-i* and *chan-i* in Data Container [14.1.3]

This subprocess seeks a match between the station codes and channel codes, symbols *sta-i* and *chan-i*, respectively, and the station codes and channel codes in the *quality-checked waveforms* container. The inline global Scheme function `search-container` is used to extract that waveform object from the waveforms container whenever both the station codes and the channel codes match the internal symbols *sta-i* and *chan-i*, respectively (see Table 224). If an exact match exists, the *wfdata* object whose attributes match is returned by the comparison code; otherwise, a null object is returned. In either case the returned object is denoted by the internal symbol *this-obj*.

### Set Attributes into *qc-stats* Object [14.1.4]

The process *Identify Missing Waveforms* writes a record to the **qcstats** database table where waveforms are missing for a requested station-channel pair. The subprocess *Set Attributes into qc-stats Object* fills in the *qc-stats* object to supply the information for this record. Therefore, in cases where the requested waveforms are missing, for the requested station-channel pair, the attributes of an initially nil *qc-stats* object are filled-in. Three attributes are set in the *qc-stats* object: station code (internal symbol *sta*), channel code (internal symbol *chan*), and time-duration of the missing waveforms (internal symbol *missing*). The global Scheme function `set-gobj-attr` is used to set these three attributes as listed in Table 225.

**TABLE 225: ATTRIBUTES SET IN QC-STATS OBJECT**

Symbol	Type	Description
<i>sta</i>	character-string	station code of this station-channel pair
<i>chan</i>	character-string	channel code of this station-channel pair
<i>missing</i>	real	time-duration of requested waveforms

## ▼ Administrative Processing

**Write Results to Database [14.1.5]**

This subprocess writes a record to the **qcstats** database table where waveforms are missing for a requested station-channel pair. It is implemented by the local Scheme function `write-qcstats`, which is called by the Scheme language code:

```
(write-qcstats qc-stats-obj)
```

The function requires one argument, which is described in Table 226.

**TABLE 226: FUNCTIONAL INTERFACE OF WRITE-QCSTATS**

Argument Name	Usage	Type	Description
<i>qc-stats-obj</i>	input and output	GObj	<i>qc-stats</i> object
n/a	returned value	logical scalar	indicates success (t) or failure (nil)

The *qc-stats* object whose symbol is *qc-stats-obj* is enhanced by setting several of its indicative attributes to appropriate values and assigning a unique identification number (internal symbol *qcstatid*) by the local Scheme function `get-qcstatsid`. This function accesses the **lastid** database table.

The function `write-qcstats` performs the following steps:

1. Create an empty container to hold the input *qc-stats* object.
2. Acquire, by access to the CVAR datastore, the “cull-list” (the list of station-channel pairs whose waveforms are deemed too defective to be used for DFX application processing during this time interval).
3. Extract the station codes and channel codes from the *qc-stats* object that was input to this function.
4. Test if this station-channel pair is found on the cull-list.
5. If this station-channel pair is not found on the cull-list, then
  - complete the input *qc-stats* object with additional information by the common *libgobj* Scheme function `set-gobj-attr`,



- insert the input *qc-stats* object into the container via the global Scheme function `insert-container`, and
  - write the input *qc-stats* object, via the global Scheme function `submit-container-db`, to the **qcstats** database table (whose identifier is acquired by access to the CVAR datastore).
6. If this station-channel pair is found on the cull-list, then do not enhance the *qc-stats* object, do not insert it into the container, and do not write it to the database.

If the input *qc-stats* object is to be written to the database, its attributes that are populated are those listed in Table 227.

**TABLE 227: ATTRIBUTES POPULATED IN QC-STATS OBJECT**

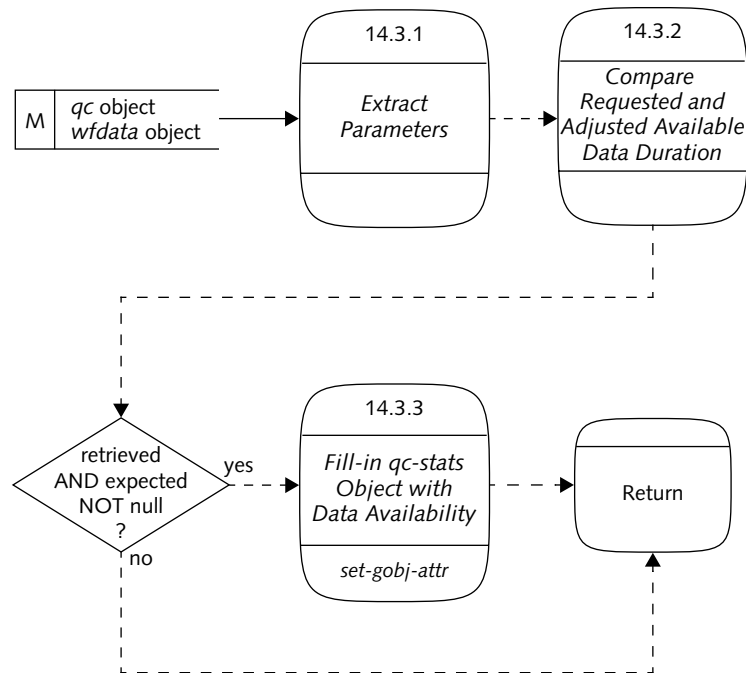
Symbol	Type	Description
<i>qcstatsid</i>	integer	unique <b>qcstats</b> table entry identification number for this station-channel pair
<i>time</i>	real	requested start-time of these waveforms
<i>endtime</i>	real	requested end-time of these waveforms
<i>jdate</i>	character-string	Julian date of start-time expressed as <i>yyyyddd</i>
<i>auth</i>	character-string	author
<i>lddate</i>	date	load date of this database insertion

### Detailed Description of Calculate Waveforms Availability [14.3]

This process compares the requested time-duration for this processing interval with the time-duration of each station-channel-pair's non-masked waveforms. It sets the *missing* attribute in the *qc-stats* object to the difference between the requested and the available time-durations. The process is implemented by a call to the Scheme-coded function `measure-data-availability`.

## ▼ Administrative Processing

Figure 51 shows the architecture of this process. The process is implemented by a single pass through three subprocesses. The first subprocess extracts parameter values for the symbols required in performing this process. The second subprocess compares the requested and available waveforms' time-durations. The third subprocess sets the *missing* attribute in the *qc-stats* object with the amount by which adjusted time-duration of the available waveforms is less than the time-duration requested.



**FIGURE 51. ARCHITECTURE OF CALCULATE WAVEFORMS AVAILABILITY**

The following subprocesses shown in Figure 51 are described in this section:

- *Extract Parameters* [14.3.1]
- *Compare Requested and Adjusted Available Data Duration* [14.3.2]
- *Fill-in qc-stats Object with Data Availability* [14.3.3]

**Extract Parameters [14.3.1]**

The process *Calculate Waveforms Availability* requires values for the requested start-times and end-times to calculate any discrepancy between the adjusted time-duration available and the time-duration requested. Subprocess *Extract Parameters* acquires the requested time-duration. The inline global Scheme function `mstpar` is used to extract values for these requested time limits via access to the CVAR datastore (see Table 228).

**TABLE 228: INTERNAL PARAMETERS OF MEASURE-DATA-AVAILABILITY**

Symbol	Type	Description
<i>t1</i>	real	requested start-time of this waveform
<i>t2</i>	real	requested end-time of this waveform
<i>retrieved</i>	real	adjusted time-duration of available waveforms

The inline local Scheme function `say-retrieved-wfdata-time` is used to extract a value for the adjusted time-duration of the available waveforms set into symbol *retrieved*.

The adjustment to the time-duration of available waveforms subtracts the time-duration of the data added at each end of the requested processing interval. Thus, if data were requested from time 10 to 24, and the data acquired spanned the interval from 7 to 26 (starting 3 earlier and running 2 later to eliminate signal processing edge effects), then the adjusted time-duration of available data would be reported as 14 (= 24 – 10). However, if data were unavailable until 13, then the data acquired would range from 13 to 26, and the adjusted time-duration of available data would be reported as 11 (= 24 – 13).

## ▼ Administrative Processing

**Compare Requested and Adjusted  
Available Data Duration [14.3.2]**

This subprocess calculates the difference between the time-duration of requested data and the adjusted time-duration of available data. The inline intrinsic Scheme code is used to ensure that neither the requested time-duration, (*expected*) = ( $t_2 - t_1$ ), nor the adjusted available time-duration, (*retrieved*), is `nil`. If both the requested and adjusted available time-durations are non-`nil`, then the inline intrinsic Scheme code is used to calculate their difference (*expected* - *retrieved*).

**Fill-in qc-stats Object with Data  
Availability [14.3.3]**

This subprocess fills in the *missing* attribute of a *qc-stats* object with the difference between the requested time-duration and the adjusted available time-duration of the waveforms. The inline common *libgobj* Scheme function `set-gobj-attr` is used to set the calculated difference in time-durations (*requested* - *adjusted available*) into the *missing* attribute in the *qc-stats* object.

## Chapter 8: Database Description

This chapter describes the database usage of *DFX* and includes the following topics:

- General Description
- Database Tables
- Output Database Attributes

## Chapter 8: Database Description

### GENERAL DESCRIPTION

*DFX* makes extensive use of the IDC relational database; however, *DFX* does not use a single set of tables. Instead each *DFX* application is configured to read and write the tables appropriate to the functionality. For certain common tasks, such as reading waveforms, there are tables used by most applications (see “Reading Waveforms” on page 58). The interface between *DFX* applications and the database is handled through the Generic Database Interface in *libgdi*.

This chapter first describes which tables are used by the 14 *DFX* applications. The remaining sections describe the specific attributes written by each application.

### DATABASE TABLES

*DFX* applications use the IDC database tables listed in Table 229. An **R** indicates that the application reads the table; a **W** indicates that the application writes the table; and a **B** indicates that the application reads and writes the table. For an application to be considered reading from a table, it must be retrieving data for some sort of processing. Most applications check, in a process known as screening, for entries that already exist in the database before writing to a table (see “Screening Database Containers” on page 43). Although the application technically reads from the table, screening is not considered a database read for the purpose of Table 229.

**TABLE 229: DATABASE TABLES USED BY EACH DFX APPLICATION**

Database Table	DFX Application	SeisDet	OriginBeam	BOTF	IntSeisRcl	AutoSeisRcl	DPSNR	NoiseAmp	SeisEvch	HydroDet	IntHydroRcl	HydroEvch	InfraDet	SegArch	QCStats
affiliation		R	R	R	R	R	R	R	R	R	R	R	R	R	
amplitude		W			W	W	W	W	W			W	W		
amp3c		W			W	W									
apma		W			W	W									
arrival		W			B	B	R	R	R	W	B		W	R	
assoc					R	R	R	R	R					R	
ceppks									W						
complexity									W						
detection		W			W	W				W	W		W		
hydro_features										W	B				
infra_features													W		
instrument		R	R	R	R	R	R	R	R	R	R	R	R	R	
lastid		B	B	B	B	B	B	B	B	B		B	B	B	B
origerr									R			R			
origin			R			R	R	R	R			R		R	
parrival								W	W			W			
qcstats		W								W			W		W
sensor		R	R	R	R	R	R	R	R	R	R	R	R	R	
site		R	R	R	R	R	R	R	R	R	R	R	R	R	
sitechan		R	R	R	R	R	R	R	R	R	R	R	R	R	
splp									W						

## ▼ Database Description

TABLE 229: DATABASE TABLES USED BY EACH DFX APPLICATION (CONTINUED)

Database Table	DFX Application	SeisDet	OriginBeam	BOTF	IntSeisRcl	AutoSeisRcl	DPSNR	NoiseAmp	SeisEvch	HydroDet	IntHydroRcl	HydroEvch	InfraDet	SegArch	QCStats
spvar									W						
timefreq									W						
thirdmom									W						
wfdisc		B	B	B	R	R	R	R	R	R	R	R	B	B	
wftag		W	W											W	

## OUTPUT DATABASE ATTRIBUTES

The following sections provide a synopsis of the attributes in each database table, which are populated by the applications. Only attributes whose values are set by the application are included. Other attributes defined in the schema but not set by the application will contain default or null values or, in the case of updating, the values already contained in the database. The following sections follow the order of the application descriptions in the previous chapters of this document.

## Seismic Processing

## Automatic Seismic Detection [1]

The *Automatic Seismic Detection* (*SeisDet*) application writes to nine database tables: **amplitude**, **amp3c**, **apma**, **arrival**, **detection**, **lastid**, **qcstats**, **wfdisc** and **wftag**. Table 230 lists the attributes populated in each table except for the **lastid** and **qcstats** tables. The **lastid** table is accessed and updated for both *arids* and *wfids* as needed. The **qcstats** table is described in “Quality Control Statistics [14]” on



page 452; in particular, see Table 243. The application may not write all of the attributes listed in Table 230 every time. The attributes written depend on the type of station being processed.

**TABLE 230: DATABASE ATTRIBUTES WRITTEN BY SEISDET**

Table	Attributes
<b>amplitude</b>	<i>ampid, arid, chan, amp, per, snr, amptime, start_time, duration, bandw, amptype, units, inarrival, auth, lddate</i>
<b>amp3c</b>	<i>arid, cfreq, vamp, vsnr, hamp, hsnr, htov, rid, lddate</i>
<b>apma</b>	<i>phase, arid, freq, snr, ampp, amps, amplr, rect, plans, planlr, hvratp, hmxmn, inang3, seazp, seazs, seazlr, inang1, ptime, stime, auth, apmarid, lddate</i>
<b>arrival</b>	<i>sta, time, arid, jdate, chan, stype, deltim, azimuth, delaz, slow, delslo, ema, rect, amp, per, snr, qual, auth, lddate</i>
<b>detection</b>	<i>arid, jdate, time, sta, chan, bmtime, cfreq, seaz, delaz, slow, delslo, snr, stav, fstat, deltim, bandw, fkqual, lddate</i>
<b>wfdisc</b>	<i>sta, chan, time wfid, jdate, endtime, nsamp, calib, calper, instype, segtype, datatype, dir, dfile, foff, lddate</i>
<b>wftag</b>	<i>tagname, tagid, wfid, lddate</i>

### Automatic Origin Beam [2]

The *Automatic Origin Beam* (*OriginBeam*) application writes to three database tables: **lastid**, **wfdisc**, and **wftag**. Table 231 lists the attributes populated in the **wfdisc** and **wftag** tables. The **lastid** table is accessed and updated for *wfids* as needed.

**TABLE 231: DATABASE ATTRIBUTES WRITTEN BY ORIGINBEAM**

Table	Attributes
<b>wfdisc</b>	<i>sta, chan, time wfid, jdate, endtime, nsamp, calib, calper, instype, segtype, datatype, dir, dfile, foff, lddate</i>
<b>wftag</b>	<i>tagname, tagid, wfid, lddate</i>

## ▼ Database Description

**Interactive Beam on the Fly [3]**

The *Interactive Beam on the Fly (BOTF)* application writes to two database tables: **lastid** and **wfdisc**. Table 232 lists the attributes populated in the **wfdisc** table. The **lastid** table is accessed and updated for *wfids* as needed.

**TABLE 232: DATABASE ATTRIBUTES WRITTEN BY BOTF**

Table	Attributes
<b>wfdisc</b>	<i>sta, chan, time wfid, jdate, endtime, nsamp, calib, calper, instype, segtype, datatype, dir, dfile, foff, lddate</i>

**Interactive Seismic Recall [4]**

The *Interactive Seismic Recall (IntSeisRcl)* application writes to six database tables: **amplitude**, **amp3c**, **apma**, **arrival**, **detection**, and **lastid**. Table 233 lists the attributes populated in each table except for the **lastid** table. The **lastid** table is accessed and updated for *ampids* as needed. The application may not write all of the attributes listed in Table 233 every time. The attributes written depend on the type of station being processed.

**TABLE 233: DATABASE ATTRIBUTES WRITTEN BY INTSEISRCL**

Table	Attributes
<b>amplitude</b>	<i>ampid, arid, chan, amp, per, snr, amptime, start_time, duration, bandw, amptype, units, inarrival, auth, lddate</i>
<b>amp3c</b>	<i>arid, cfreq, vamp, vsnr, hamp, hsnr, htov, rid, lddate</i>
<b>apma</b>	<i>phase, arid, freq, snr, ampp, amps, amplr, rect, plans, planlr, hvratp, hmxmn, inang3, seazp, seazs, seazlr, inang1, ptime, stime, auth, apmarid, lddate</i>
<b>arrival</b>	<i>chan, channel, deltim, azimuth, delaz, slow, delslo, ema, rect, amp, per, logat, snr, qual, auth</i>
<b>detection</b>	<i>arid, jdate, time, sta, chan, bmtime, cfreq, seaz, delaz, slow, delslo, snr, stav, fstat, deltim, bandw, fkqual, lddate</i>

### Automatic Seismic Recall [5]

The *Automatic Seismic Recall (AutoSeisRcl)* application writes to six database tables: **amplitude**, **amp3c**, **apma**, **arrival**, **detection**, and **lastid**. Table 234 lists the attributes populated in each table except for the **lastid** table. The **lastid** table is accessed and updated for *ampids* as needed. The application may not write all of the attributes listed in Table 234 every time. The attributes written depend on the type of station being processed.

**TABLE 234: DATABASE ATTRIBUTES WRITTEN BY AUTOSEISRCL**

Table	Attributes
<b>amplitude</b>	<i>ampid, arid, chan, amp, per, snr, amptime, start_time, duration, bandw, amptype, units, inarrival, auth, lddate</i>
<b>amp3c</b>	<i>arid, cfreq, vamp, vsnr, hamp, hsnr, htov, rid, lddate</i>
<b>apma</b>	<i>phase, arid, freq, snr, ampp, amps, amplr, rect, plans, planlr, hvratp, hmxmn, inang3, seazp, seazs, seazlr, inang1, ptime, stime, auth, apmarid, lddate</i>
<b>arrival</b>	<i>chan, channel, deltim, azimuth, delaz, slow, delslo, ema, rect, amp, per, logat, snr, qual, auth</i>
<b>detection</b>	<i>arid, jdate, time, sta, chan, bmtyp, cfreq, seaz, delaz, slow, delslo, snr, stav, fstat, deltim, bandw, fkqual, lddate</i>

### Automatic Depth-phase SNR [6]

The *Automatic Depth-phase SNR (DPSNR)* application writes to two database tables: **amplitude** and **lastid**. Table 235 lists the attributes populated in the **amplitude** table. The **lastid** table is accessed and updated for *ampids* as needed.

**TABLE 235: DATABASE ATTRIBUTES WRITTEN BY DPSNR**

Table	Attributes
<b>amplitude</b>	<i>ampid, arid, chan, amp, per, snr, amptime, start_time, duration, bandw, amptype, units, inarrival, auth, lddate</i>

## ▼ Database Description

**Automatic Noise Amplitude Estimation [7]**

The *Automatic Noise Amplitude Estimation* (NoiseAmp) application writes to three database tables: **amplitude**, **lastid**, and **parrival**. Table 236 lists the attributes populated in the **amplitude** and **parrival** tables. The **lastid** table is accessed and updated for both *ampids* and *parids* as needed.

**TABLE 236: DATABASE ATTRIBUTES WRITTEN BY NOISEAMP**

Table	Attributes
<b>amplitude</b>	<i>ampid, parid, chan, amp, per, snr, amptime, start_time, duration, bandw, amptype, units, inarrival, auth, lddate</i>
<b>parrival</b>	<i>parid, orid, evid, sta, time, azimuth, slow, phase, delta, vmodel, lddate</i>

**Automatic Seismic Event Characterization [8]**

The *Automatic Seismic Event Characterization* (SeisEvch) application writes to nine database tables: **amplitude**, **ceppks**, **complexity**, **lastid**, **parrival**, **splp**, **spvar**, **timefreq**, and **thirdmom**. Table 237 lists the attributes populated in each table except the **lastid** table. The **lastid** table is accessed and updated for both *ampids* and *parids* as needed.

**TABLE 237: DATABASE ATTRIBUTES WRITTEN BY SEISEVCH**

Table	Attributes
<b>amplitude</b>	<i>ampid, parid, chan, amp, per, snr, amptime, start_time, duration, bandw, amptype, units, inarrival, auth, lddate</i>
<b>ceppks</b>	<i>orid, sta, ptyp, pkamp, pkqf, lddate</i>
<b>complexity</b>	<i>orid, sta, phase, rectype, complexity, snr, lddate</i>
<b>parrival</b>	<i>parid, orid, evid, sta, time, azimuth, slow, phase, delta, vmodel, lddate</i>
<b>splp</b>	<i>orid, sta, rectype, ratio, lddate</i>

TABLE 237: DATABASE ATTRIBUTES WRITTEN BY SEISEVCH (CONTINUED)

Table	Attributes
<b>spvar</b>	<i>arid, fsid, acoef, bcoef, ccoef, fmin, fmax, svar, lddate</i>
<b>thirdmom</b>	<i>arid, sta, rectype, tmf, tmfpct, lddate</i>
<b>timefreq</b>	<i>orid, sta, rectype, zavpct, navpct, eavpct, zavcep, navcep, eavcep, zavcor, navcor, eavcor, xcor, lddate</i>

## Hydroacoustic Processing

### Automatic Hydroacoustic Detection [9]

The *Automatic Hydroacoustic Detection (HydroDet)* application writes to five database tables: **arrival**, **detection**, **hydro\_features**, **lastid**, and **qcstats**. Table 238 lists the attributes populated in each table except for the **lastid** and **qcstats** tables. The **lastid** table is accessed and updated for *arids* as needed. The **qcstats** table is described in “Quality Control Statistics [14]” on page 452; in particular, see Table 243.

TABLE 238: DATABASE ATTRIBUTES WRITTEN BY HYDRODET

Table	Attributes
<b>arrival</b>	<i>sta, time, arid, jdate, chan, deltim, snr, auth, lddate</i>
<b>detection</b>	<i>arid, jdate, time, sta, chan, bmtime, snr, stav, deltim, lddate</i>
<b>hydro_features</b>	<i>arid, peak_time, peak_level, total_energy, mean_arrival_time, time_spread, onset_time, termination_time, total_time, num_cross, ave_noise, skewness, kurtosis, cep_var_signal, cep_delay_time_signal, cep_peak_std_signal, cep_var_trend, cep_delay_time_trend, cep_peak_std_trend, low_cut, high_cut, ford, ftype, fzp, prob_weight_time, sigma_time, lddate</i>

## ▼ Database Description

**Interactive Hydroacoustic Recall [10]**

The *Interactive Hydroacoustic Recall (IntHydroRcl)* application writes to three database tables: **arrival**, **detection**, and **hydro\_features**. Table 239 lists the attributes populated in each table. For **arrival** and **detection** it updates only *time* and *deltim*. For **hydro\_features** it updates all but onset and termination times and the filter parameters.

**TABLE 239: DATABASE ATTRIBUTES WRITTEN BY INTHYDRORCL**

Table	Attributes
<b>arrival</b>	<i>time, deltim, lddate</i>
<b>detection</b>	<i>time, deltim, lddate</i>
<b>hydro_features</b>	<i>peak_time, peak_level, total_energy, mean_arrival_time, time_spread, total_time, num_cross, ave_noise, skewness, kurtosis, cep_var_signal, cep_delay_time_signal, cep_peak_std_signal, cep_var_trend, cep_delay_time_trend, cep_peak_std_trend, prob_weight_time, sigma_time, lddate</i>

**Automatic Hydroacoustic Event Characterization [11]**

The *Automatic Hydroacoustic Event Characterization (HydroEvch)* application writes to two database tables: **amplitude** and **parrival**. Table 240 lists the attributes populated in each table.

**TABLE 240: DATABASE ATTRIBUTES WRITTEN BY HYDROEVCH**

Table	Attributes
<b>amplitude</b>	<i>ampid, parid, chan, amp, per, snr, amptime, start_time, duration, bandw, amptype, units, auth, lddate</i>
<b>parrival</b>	<i>parid, orid, evid, sta, time, azimuth, slow, phase, delta, vmodel, lddate</i>

## Infrasonic Processing

### Automatic Infrasonic Detection [12]

The *Automatic Infrasonic Detection (InfraDet)* application writes to seven database tables: **amplitude**, **arrival**, **detection**, **infra\_features**, **lastid**, **qcstats**, and **wfdisc**. Table 241 lists the attributes populated in each table except for the **lastid** and **qcstats** tables. The **lastid** table is accessed and updated for *ampids*, *arids* and *wfids* as needed. The **qcstats** table is described in “Quality Control Statistics [14]” on page 452; in particular, see Table 243 on page 452.

**TABLE 241: DATABASE ATTRIBUTES WRITTEN BY INFRADET**

Table	Attributes
<b>amplitude</b>	<i>ampid, arid, chan, amp, per, snr, amptime, start_time, duration, bandw, amptype, units, inarrival, auth, lddate</i>
<b>arrival</b>	<i>sta, time, arid, jdate, chan, deltim, azimuth, delaz, slow, delslo, amp, per, snr, qual, auth, lddate</i>
<b>detection</b>	<i>arid, jdate, time, sta, chan, bmtime, cfreq, seaz, delaz, slow, delslo, snr, stav, fstat, deltim, bandw, fkqual, lddate</i>
<b>infra_features</b>	<i>arid, eng_time, eng_dur, eng_deldur, coh_time, coh_dur, coh_deldur, coinc_time, coinc_dur, coinc_deldur, ford, zrcr_freq, zrcr_delfreq, crnr_freq, crnr_delfreq, coh_pre, coh_snr, total_energy, auth, lddate</i>
<b>wfdisc</b>	<i>sta, chan, time wfid, jdate, endtime, nsamp, calib, calper, instype, seg-type, datatype, dir, dfile, foff, lddate</i>

## Administrative Processing

### Segment Archiving [13]

The *Segment Archiving (SegArch)* application writes to three database tables: **wfdisc**, **wftag**, and **lastid**. Table 242 lists the attributes populated in the **wfdisc** and **wftag** tables. The **lastid** table is accessed and updated for *wfids* as needed.

## ▼ Database Description

TABLE 242: DATABASE ATTRIBUTES WRITTEN BY SEGARCH

Table	Attributes
wfdisc	<i>sta, chan, time wfid, jdate, endtime, nsamp, calib, calper, instype, segtype, datatype, dir, dfile, foff, lddate</i>
wftag	<i>tagname, tagid, wfid, lddate</i>

**Quality Control Statistics [14]**

The *Quality Control Statistics* (QCStats) application writes to two database tables: **qcstats** and **lastid**. Table 243 lists the attributes populated in the **qcstats** table. The **lastid** table is accessed and updated for *qcstatsids* as needed.

TABLE 243: DATABASE ATTRIBUTES WRITTEN BY QCSTATS

Table	Attributes
qcstats	<i>qcstatsid, sta, chan, time, jdate, endtime, dettime, detendtime, missing, dropped, nseg, masked, pointspike, spike, nconstseg, const, avgconstval, stdconstval, auth, lddate</i>



## References

The following sources supplement or are referenced in document:

- [Abe96] Abelson, H., Sussman, G. J., and Sussman, J., *Structure and Interpretation of Computer Programs*, 2nd Edition, MIT Press, Cambridge, MA, 1996.
- [Fri97] Friedman, D., and Felleisen, M., *The Little Schemer*, MIT Press, 1997.
- [Gan79] Gane, C., and Sarson, T., *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.
- [IDC5.1.1Rev2] Science Applications International Corporation, Veridian Pacific-Sierra Research, *Database Schema, Revision 2*, SAIC-00/3057, PSR-00/TN2830, 2000.
- [IDC5.1.2] Science Applications International Corporation, Pacific-Sierra Research Corporation, *Database Tutorial*, SAIC-99/3022, PSR-99/TN1145, 1999.
- [IDC5.1.3Rev0.1] Science Applications International Corporation, Veridian Pacific-Sierra Research, *Configuration of PIDC Databases*, SAIC-01/3022, PSR-99/TN1114, 2001.
- [IDC5.2.1] Science Applications International Corporation, *IDC Processing of Seismic, Hydroacoustic, and Infrasonic Data*, SAIC-99/3023, 1999.
- [IDC6.2.4] Science Applications International Corporation, *Configuration of PIDC Processing Data Files*, SAIC-99/3025, 1999.

## ▼ References

- [Lan97g] Laney, H., *Hydroacoustic Feature Extractor (DFX-H Version 1) Technical Description*, CMR-97/06, 1997.
- [WGB00c] Working Group B, *Proceedings of IDC Technical Experts Meeting on Seismic-Acoustic Event Screening*, Preparatory Commission of the Comprehensive Nuclear Test-Ban Treaty Organization, CTBT/WGB/TL-2/58, 2000.

# Glossary

Selected terms in this glossary are specific to this document and are not applicable in any other context. These glossary items are indicated by appending “(DFX only)” to the term.

## Symbols

### 3-C

Three-component.

### μPa

MicroPascals.

## A

### abs

Absolute value.

### AIC

Akaike Information Criterion. A mathematical method for distinguishing between noise and signal. Used in estimating onset times. See [IDC5.2.1]

### Alpha List

(ARS) List of stations and phases contributing to a S/H/I event location.

### analyst

Personnel responsible for reviewing and revising the results of automatic processing.

### API

Application Program Interface.

### AR

Auto-Regressive. A predictive model used to estimate current parameter values based upon earlier data values.

### ARAIC

Auto-Regressive Akaike Information Criterion. A method that uses an auto-regressive model to implement the Akaike Information Criterion. See AIC.

### array

Collection of sensors distributed over a finite area (usually in a cross, triangle, or concentric pattern) and referred to as a single station.

### arrival

Detected signal that has been associated to an event. First, the Global Association (GA) software associates the detection to an event. Later, during interactive processing, many arrivals are confirmed, improved, or added by visual inspection.

## ▼ Glossary

**ARS**

Analyst Review Station. This application provides tools for an analyst to refine and improve the event bulletin by interactive analysis.

**arsoc (DFX only)**

Generic object in *DFX* that combines arrival and association information together.

**ASCII**

American Standard Code for Information Interchange. Standard, unformatted 256-character set of letters and numbers.

**associated phase**

Phase that is associated with an S/H/I event.

**attribute**

(1) Database column. (2) Characteristic of an item; specifically, a quantitative measure of a S/H/I detection such as azimuth, slowness, period, or amplitude.

**AutoSeisRcl**

Automatic Seismic Recall processing. A *DFX* Scheme application run in post-analysis processing.

**azimuth**

Direction, in degrees clockwise with respect to North, from a station to an event.

**B****back azimuth**

Direction, in degrees, from an event to a station.

**beam**

(1) Waveform created from array station elements that are sequentially summed after being steered to the direction of a specified azimuth and slowness. (2) Any derived waveform (for example, a filtered waveform).

**beam recipe**

Table of parameters that describe how to create and use a beam. Parameters may include the channels to use in forming the beam, the filter to use, the direction and slowness in which to steer the beam, and the snr threshold to apply for detection.

**Beamer**

Application that prepares origin beams for interactive analysis.

**beamform**

Sum a set of waveforms from array station elements with time delays introduced to compensate for the time it takes a wave to travel across the array.

**best beam**

Beam, in the set of all detection beams, with the highest signal-to-noise ratio for an arrival.

**blockage (hydroacoustic)**

Obstruction in the direct oceanic path between the source and the receiver.

**BOTF**

Beam-on-the-Fly processing. A *DFX* Scheme application that creates origin beams and can be called from within *ARS*.

**bulletin**

Chronological listing of event origins spanning an interval of time. Often, the specification of each origin or event is accompanied by the event's arrivals and sometimes with the event's waveforms.

**C****CDS** (*DFX* only)

See Coincidence Detection Space.

**cepstrum** (*cepstral*)

Fourier transformation of a power spectrum whose magnitudes have been scaled logarithmically.

**chan.**

Channel.

**channel**

Component of motion or distinct stream of data.

**circular file**

Fixed-length data file constructed such that, once full of data, new data overwrite the oldest data.

**class**

Defined structure for a set of similar objects that share common specified characteristics, including data items and operations.

**cluster** (*DFX* only)

Group of candidate detections with similar arrival angles and slowness values. Clusters are created from the Primary Detection Space of the infrasonic detector.

**CMR**

Center for Monitoring Research.

**coherence detector**

Process that declares a transient signal detection when the spatial coherence across an array rises above a given threshold.

**coherence trace** (*DFX* only)

One of three waveforms derived from the spatial coherence level observed in the Primary Detection Space of the infrasonic detector.

**coherent beam**

Beam formed from unrectified waveform data (usually zero-measured), which utilizes phase information in the waveform to enhance coherent energy at the azimuth and slowness of the beam and to suppress random noise.

**Coincidence Detection Space** (*DFX* only)

Space that is used in infrasonic detection processing wherein final detections are determined from candidate detections created in the Primary Detection Space. Both spatial coherence and energy level are considered in determining the final detections.

## ▼ Glossary

**coincidence detector**

Detector that requires both the signal energy and the spatial coherence to rise above thresholds in the same time window.

**commit**

Process of saving changes made to the database.

**complexity**

Measure of the teleseismic P phase that compares energy in the initial signal with energy in the coda. This measure can be used in event characterization.

**Comprehensive Nuclear-Test-Ban Treaty Organization**

Treaty User group that consists of the Conference of States Parties, the Executive Council, and the Technical Secretariat.

**Computer Software Component**

Functionally or logically distinct part of a computer software configuration item; possibly an aggregate of two or more software units.

**Computer Software Configuration Item**

Aggregation of software that is designated for configuration management and treated as a single entity in the configuration management process.

**configuration**

(1) (hardware) Arrangement of a computer system or components as defined by the number, nature, and interconnection of its parts. (2) (software) Set of

adjustable parameters, usually stored in files, which control the behavior of applications at run time.

**container**

Type of generic object used to hold other objects (similar to an array). All of the objects in the container must be of the same type.

**continuous waveform data**

Waveform data that are transmitted to the IDC on a nominally continuous basis.

**control flow**

Sequence in which operations are performed during the execution of a computer program.

**COTS**

Commercial-Off-the-Shelf; terminology that designates products such as hardware or software that can be acquired from existing inventory and used without modification.

**CSC**

Comprehensive Nuclear-Test-Ban Treaty Organization.

**CSCI**

Computer Software Configuration Item.

**CTBTO**

Comprehensive Nuclear-Test-Ban Treaty Organization.

**CVAR datastore**

C variable data storage area (sometimes referred to as the CVAR table) that stores name-value pairs usually provided

by application configuration files and command line arguments. The values are available through “C” and the Scheme API.

## D

### DACS

Distributed Application Control System. This software supports inter-application message passing and process management.

### dB

Decibel.

### db<tablename> (DFX only)

Generic object with fields that have a one-to-one correspondence with the database table attributes.

### depth phase

Seismic signal that travels upwards from the hypocenter, reflects off the earth's surface, and travels the remaining path as a typical refracted phase. It is usually denoted by a phase name starting with a lower case “p” or “s” that indicates the upward leg of the path.

### detection

Probable signal that has been automatically detected by the *Detection and Feature Extraction (DFX)* software.

### detection beam

Same as a f-k beam.

### detection object (or det)

Type of generic object that holds the working attributes of a detected signal.

### DFX

Detection and Feature Extraction. *DFX* is a programming environment that executes applications written in Scheme (known as *DFX* applications).

### DFX/lib<name> function (DFX only)

Scheme symbol that is bound to a C function in the named *DFX* library. The function is then available from the Scheme interface like any other Scheme function.

### DLMan

Disk Loop Manager. An IDC application in the Data Services CSCI.

### DPSNR

Automatic Depth-Phase Signal-to-Noise Ratio processing. A *DFX* Scheme application that makes a special snr measurement of depth-phase arrivals.

## E

### ema

Emergence angle.

### epoch time

Number of seconds after January 1, 1970 00:00:00.0.

### event

Unique source of seismic, hydroacoustic, or infrasonic wave energy that is limited in both time and space.

## ▼ Glossary

**event characterization**

IDC process of characterizing events by features of signals recorded at one or more stations.

**event screening**

IDC process that produces the SSEB by removing events of clear natural origin from the SEB.

**F**
**f-k beam**

Coherent beam steered to the azimuth and slowness of the maximum peak in an f-k spectrum.

**f-k spectrum**

Result of a data transformation from the time-space domain to the frequency-wavenumber domain. Useful in determining the direction and slowness of an arriving phase.

**F-statistic**

Measure that indicates the degree of spatial coherence of a waveform across an array of sensors. This measure is approximately equal to the ratio of the spatially coherent energy to the incoherent energy scaled by the number of non-collocated sensors.

**features**

Various measurements of a waveform segment used to characterize a detection.

**field**

- (1) Attribute of a generic object.
- (2) Attribute in a database table (the name of the column).

**FIR**

Finite Impulse Response (usually in reference to a filter).

**G**
**GA**

Global Association application. GA associates S/H/I phases to events.

**garbage collect**

Background activity of searching memory used for dynamically allocated data to find areas that are no longer in use and free them. This is performed automatically in Scheme and other Lisp dialects.

**gc**

See garbage collect.

**generic object**

Construct used to hold and manipulate data. The type of object determines the data that it can contain. Also known as an object or GObj.

**global Scheme function**

Scheme function that is available to all DFX applications and is defined in the `DFXdefault.scm` file.

**GObj**

See generic object.



## H

### hydroacoustic

Pertaining to sound in the ocean.

### HydroDet

Automatic Hydroacoustic Detection processing. A *DFX* Scheme application that makes detections and measures features on hydroacoustic waveform data.

### HydroEvch

Automatic Hydroacoustic Event Characterization processing. A *DFX* Scheme application that computes hydroacoustic measures relative to events typically formed by the seismic network. It can be used to rule out the possibility that the event occurred in water.

### Hz

Hertz.

## I

### IDC

International Data Centre.

### IIR

Infinite Impulse Response (filters also referred to as recursive filters).

### IMS

International Monitoring System.

### in-line code

Set of source code that performs a specific task but is not organized as an independent function.

### incidence angle

Angle relative to vertical that a wave is travelling when it reaches the receiver.

### InfraDet

Automatic Infrasonic Detection processing. A *DFX* Scheme application that makes detections and measures features on infrasonic waveform data.

### infrasonic (infrasound)

Pertaining to low-frequency (sub-audible) sound in the atmosphere.

### initsite (*DFX* only)

Class of generic objects that contain site-specific data, in particular recipe parameters. The initsite objects are specialized to each *DFX* Scheme application.

### IntHydroRcl

Interactive Hydroacoustic Recall processing. A *DFX* Scheme application that is called from *ARS* to measure features on analyst modified or added hydroacoustic detections.

### intrinsic Scheme function

Function that is part of the original Scheme language or *libscheme*.

### IntSeisRcl

Interactive Seismic Recall processing. A *DFX* Scheme application that is called from *ARS* to measure features on analyst modified or added seismic detections.

## ▼ Glossary

**IPC**

Interprocess communication. The messaging system by which applications communicate with each other through *libipc* common library functions. See *tuxshell*.

**J****jdate**

Modified Julian Date. Concatenation of the year and three-digit Julian day of year. For example, the *jdate* for 07 March, 2000, is 2000067.

**L****larray**

Large array.

**LISP**

List Processing programming language.

**local Scheme function**

Scheme function that is defined within an application's source code file.

**LR**

Rayleigh wave. A seismic phase that travels along the surface of the earth.

**LTA (or LTAV)**

Long Term Average. A running average of the absolute value or squared value of a waveform. The averaging window is long compared to the short-term averaging window.

**M****magnitude**

Empirical measure of the size of an event (usually made on a log scale).

**magnitude slowness**

Magnitude of the slowness vector.

**Map**

Application for displaying S/H/I events, stations, and other information on geographical maps.

**mask**

Array of start and end indices for defective data samples in a time series used to later identify the defective data.

 **$m_b$** 

Magnitude estimated from seismic body waves.

**mHz**

Millihertz. One thousandth of one cycle (occurrences, alterations, pulses) per second.

 **$M_L$** 

Magnitude estimated from seismic waves measured near the source.

**N****ncc (DFX only)**

Abbreviation for the normalized cross-correlation.

**nil**

(1) Empty set. (2) Specific value in Scheme that, in a conditional statement, evaluates to false.

**NoiseAmp**

Automatic Noise Amplitude Estimation. A *DFX* Scheme application that measures the noise level at stations that did not detect signals from a given event.

**normalized cross-correlation**

Value of a cross-correlation coefficient in which the underlying cross-correlation function between two zero-mean processes, *x* and *y*, has been normalized by the square root of the product of the alignment-dependent variances in *x* and *y*.

**O****object**

Same as a generic object.

**origin**

Hypothesized time and location of a seismic, hydroacoustic, or infrasonic event. An event may have many origins. Characteristics such as magnitudes and error estimates may be associated with an origin.

**origin beam**

Coherent beam steered to the estimated event origin.

**OriginBeam**

Automatic Origin Beam processing. A *DFX* Scheme application that computes and saves origin beams for use in interactive analysis.

**P****P phase**

Seismic wave that travels from the event to the station as a compressional wave through the solid earth.

**par**

See parameter.

**parameter**

User-specified token that controls some aspect of an application (for example, database name, threshold value). Most parameters are specified using [*token* = *value*] strings, for example, `dbname=mydata/base@oracle`.

**parrival**

Database table that contains the predicted arrivals and associations for origin-based amplitude measurements.

**PcP phase**

Seismic compressional wave that reflects off the core-mantle boundary.

**PDS (*DFX* only)**

See Primary Detection Space.

**phase**

Arrival that is identified based on its path through the earth.

## ▼ Glossary

**picket**

(1) Sample in a discrete frequency spectrum. (2) Point in a discretely sampled two-dimensional slowness-plane.

**PIDC**

Prototype International Data Centre.

**pipeline**

Flow of data at the IDC from the receipt of communications to the final automated processed data before analyst review.

**PKP phase**

Seismic compressional wave that travels from the event to the station as a compressional wave through the crust, mantle, and outer core.

**polarization analysis**

Analysis to determine the propagation vector that describes particle motion. Used at seismic 3-C stations.

**post-analysis processing**

Automated processing that occurs after analysts have reviewed the automatic event bulletins.

**post-location processing**

Software that computes various magnitude estimates and selects data to be retrieved from auxiliary stations.

**Primary Detection Space (DFX only)**

Six-dimensional space used in infrasonic detection processing to hold candidate detections. The six dimensions of this space are: time (processing window time), frequency (bandpass filtering

applied to the waveforms), space (group of sensors), arrival azimuth, magnitude slowness, and spatial coherence level.

**primary stations**

Stations that make up the primary seismic network of the IMS. Primary stations send data continuously to the IDC.

**process**

Function or set of functions in an application that perform a task.

**product**

Bulletins, data, and other information collected, produced, and distributed by the IDC.

**Q**
**QC**

Quality Control.

**QCStats**

Quality Control Statistics. A DFX Scheme application that computes and saves a variety of statistics related to the input waveform data quality.

**quefrency**

Time-delay axis with units of seconds for a cepstrum.

**query**

Request for specific data from a database.

## R

### raw waveform

Waveform data as they are received from a station.

### RDBMS

Relational Database Management System.

### REB

Reviewed Event Bulletin; the bulletin formed of all S/H/I events that have passed analyst inspection and quality assurance review.

### recall processing

Type of *DFX* processing that updates arrival features based on changes made during analyst review.

### recipe

Collection of related parameters used in performing a specific task. Recipe parameters are usually defined in par files.

### regional

(1) (distance) Source-to-seismometer separations between a few degrees and 20 degrees. (2) (event) Recorded at distances where the first P and S waves from shallow events have traveled along paths through the uppermost mantle.

### rms

Root mean square.

### rollback

Process of returning a database to its state at the last commit to undo erroneous updates.

## S

### sbcc (*DFX* only)

Scheme symbol for a container of score board containers used in infrasonic detection processing.

### sbsnr

Signal-to-noise ratio measured on a standard beam.

### schema

Database structure description.

### Scheme

Dialect of the Lisp programming language that is used to configure some IDC software.

### screen container

Remove objects from a container because they either do not meet some criteria or they already exist in the database.

### seaz

Station-to-event azimuth. Usually predicted from the origin location except in the **detection** table where it is estimated from the data.

### SEB

Standard Event Bulletin; a list of analyst reviewed S/H/I events and event parameters (origin and associated arrival infor-

## ▼ Glossary

mation). The SEB is similar to the REB, but also includes event characterization parameters and event screening results for each event.

**SegArch**

Segment Archiving. A *DFX* Scheme application that extracts, beamforms, and stores waveform data relevant to an event (occurs after analyst review).

**SeisDet**

Automatic Seismic Detection processing. A *DFX* Scheme application that makes detections and measures features on seismic waveform data.

**SeisEvch**

Automatic Seismic Event Characterization processing. A *DFX* Scheme application that makes measurements on seismic waveforms in a time window predicted to contain signals from an event in order to characterize the event.

**seismic**

Pertaining to elastic waves traveling through the earth.

**side effect (DFX only)**

Change in value of a function parameter or global variable by a function without explicitly returning the modified value. Side effects in *DFX* are most often modifications to input arguments of a function.

**SIOD**

Scheme In One Defun (version of Scheme).

**site**

Location of a sensor within a station.

**slowness**

Inverse of velocity, in seconds/degree; a large slowness has a low velocity.

**slowness vector**

Vector in 2-D wavenumber space. The magnitude of the vector corresponds to the inverse of the phase velocity of a traveling plane wave. The direction of the vector is usually defined as being from the station to the source.

**slplcc (DFX only)**

Scheme symbol for container of slowness plane containers (used in infrasonic detection processing).

**SMULT analysis (DFX only)**

Source Multiplicity analysis. A method to find evidence in a signal for a multiple source event such as ripple fire from a mining explosion.

**snr**

Signal-to-noise ratio.

**spectrum (spectral)**

Plot of the energy contained in waveforms as a function of frequency.

**splp**

Short-period to long-period amplitude ratio used in event characterization.

**spvar**

Spectral Variance. A database table that contains information about the detrended log spectrum of a signal.

**SQL**

Structured Query Language; a language for manipulating data in a relational database.

**SSEB**

Standard Screened Event Bulletin; similar in content and format to the Standard Event Bulletin (SEB), but does not include events that were screened out by a standard set of event screening criteria.

**sta**

Station.

**STA (or STAV)**

Short-term average. A running average of the absolute value or squared value of a waveform. The averaging window is short in duration compared to the LTA.

**STA/LTA detector**

Process that declares a detection of a transient signal when the ratio between the STA and LTA time series rises above a given threshold.

**standard beam**

Beam in which the delays between stations are based on a specific pre-defined set of recipe parameters (referred to as the standard recipe parameters).

**StaPro**

Station Processing application for S/H/I data.

**station**

Collection of one or more monitoring instruments. Stations can have either one sensor location (for example, BGCA) or a spatially distributed array of sensors (for example, ASAR).

**steer**

Construct a beam using time delays consistent with a particular azimuth and slowness.

**subprocess**

Function or set of functions that perform a task subordinate to a process.

**T****teleseismic**

1) (distance) Source-to-seismometer separations of 20 degrees or more. (2) (event) Recorded at distances where the first P and S waves from shallow events have traveled paths through the mantle/core.

**TF (DFX only)**

Time Frequency analysis.

**third moment of frequency**

Measure of the frequency content of a signal. Used in event characterization.

**TI recipe**

Time Interval recipe. Parameters used by DFX to specify a time interval of data to read or process.

## ▼ Glossary

**time series**

Time ordered sequence of data samples. Typically a waveform or derived from waveforms, such as a beam.

**TM**

Threshold monitoring. A technique used to track the minimum detectable event size based on noise levels at stations in the detecting network.

**TMF**

See third moment of frequency.

**transient signal**

Packet of energy of finite duration that stands out from the background ambient noise.

**trigger**

Object used in *DFX* to indicate a potential transient signal for a given beam. Triggers for a set of beams are collected to form detections.

**tuple**

Database record (a row of a table).

**tuxshell**

Process in the Distributed Processing CSCI used to execute and manage applications. See IPC.

**U****UNIX**

Trade name of the operating system used by the Sun workstations.

**V****vector slowness**

Magnitude slowness with a direction.

**vernier f-k spectrum (*DFX* only)**

F-k spectrum generated by pre-aligning waveforms to a vector slowness; it is used to refine an initial coarse estimate. The vernier f-k spectrum covers a much smaller area in wavenumber space at a higher resolution to provide an improved slowness estimate.

**W****waveform**

Time-domain signal data from a sensor (the voltage output) where the voltage has been converted to a digital count (which is monotonic with the amplitude of the stimulus to which the sensor responds).

**wfdata object**

Type of generic object that holds waveforms and site related information. The waveforms are actually contained in wfmem objects that are referenced in the wfdata objects.

**wfmem object**

Type of generic object that holds time series data and the corresponding **wfdisc** database entries. Used in conjunction with wfdata objects.



**workstation**

High-end, powerful desktop computer preferred for graphics and usually networked.

**wrapper function**

Function used to provide an alternative interface to another function.



# Index

## A

administrative processing 36, 405, 406  
**affiliation** 28  
**amp3c** 28, 445, 446, 447  
**amplitude** 28, 445, 446, 447, 448, 450, 451  
*Analyst\_Log* 7  
**apma** 28, 445, 446, 447  
 application par file 25  
**arrival** 29, 445, 446, 447, 449, 450, 451  
 ARS 6, 12  
**assoc** 29  
*Automatic Depth-Phase SNR* 7, 218  
   description 10  
*Automatic Hydroacoustic Detection* 6, 290  
   description 10  
*Automatic Hydroacoustic Event Characterization* 8, 323  
   description 10  
*Automatic Infrasonic Detection* 6, 340  
   description 10  
*Automatic Noise Amplitude Estimation* 7, 232  
   description 10  
*Automatic Origin Beam* 6  
   description 10  
   *OriginBeam* 148  
*Automatic Seismic Detection* 6, 80  
   description 10  
*Automatic Seismic Event Characterization* 7, 249

description 10

*AutoSeisRcl* 7, 202  
   architecture 204  
   database attributes written 447  
   data updated 207  
   description 10  
   *Initialize for Residual Update* 210  
   *Initialize Site* 211  
   input parameters 205  
   *Perform Recall Signal Processing* 212, 213  
     *Acquire Waveforms* 214  
     architecture 214  
     *Create Detection* 215  
     *Estimate Arrival Features and Write to Database* 217  
     *Identify Best Beam* 216  
     *Revise Detection Attributes* 216  
     *Screen Detections* 216  
   *Query for Arrivals* 208

## B

*Beamer* 12  
*beam* object 72  
*beam-rec* object 72  
 BOTF 164  
   architecture 165  
   database attributes written 446  
   data produced 168  
   *Extract Parameters* 168  
   *Generate and Write Origin Beam* 170, 178  
     *Acquire Waveforms* 180  
     architecture 179  
     *Check Quality of Waveforms* 180  
     *Compute Time Intervals* 180  
     *Create Beam* 181

▼ Index

*Write to Database* 182  
*Initialize Sites* 169, 170  
     architecture 172  
     *Compute Time Intervals* 175  
     *Create initsite* 173  
     *Create Waveform Containers* 176  
     *Populate initsite Container* 177  
     *Query for Sites* 173  
     *Read Travel-time Tables* 174  
 input parameters 166

## C

catch 76  
**ceppks** 29, 448  
 check-arrival-depth-phase 195  
**complexity** 29, 448  
 compute-complexity 285  
 compute-det-amp-best-beam 127  
 compute-origin-beams 161, 181, 420  
 compute-site-origin-ti 180, 417  
     functional interface 64  
 compute-snr 105, 118, 306  
 compute-splp 281  
 compute-stations-ti 175  
 compute-tf 277  
 compute-ti-parameters 418  
 compute-tmf 279  
 container object 21  
 containers  
     internal screening 74  
     looping over contents 74  
 conventions  
     data flow symbols vi  
     typographical vii  
 create-dbamp3c-from-  
     detection 146  
 create-dbapma-from-detection 137  
 create-db-detection-from-  
     detection 142  
 create-depth-phase-initsite 224  
 create-detection-initsite 81, 88

create-gobj 71, 111  
 create-hydro-detection-  
     initsite 295  
 create-hydro-recall-initsite 317  
 create-keyed-list-container 73  
 create-recall-det-from-  
     dbarrival 215  
 create-recall-initsite 190, 211  
 create-sbsnr-dbamplitude-from-  
     det 144  
 create-secondary-beam-element-  
     object 395  
 create-secondary-beam-recipe-  
     container 396  
 create-station-event-beam 170, 178  
 CVAR 16  
 CVAR datastore 20  
     setting and retrieving parameters 75

## D

DACS 6, 27, 38  
 database  
     opening and closing 41  
     querying 41  
     screening containers 43  
     writing 44  
 database description 441, 442  
 database objects 42  
*db<table>* object 72  
*dbconnection* object 72  
 define-gobj-class 71  
 delete-<name>-recipe-cvar 76  
 delete-redundant-det-objects 386  
 depart-with-grace 403  
**detection** 29, 445, 446, 447, 449, 450, 451  
*detection* object 72  
 DFX  
     administrative processing 36, 405, 406  
     application abbreviations 11  
     command line 25  
     database operations 40

- database tables used 443
  - functional description 30
  - functionality 8
  - functions 22
  - hydroacoustic processing 34, 290
  - infrasonic processing 36, 339, 340
  - library functions 22
  - Scheme files 11
  - seismic processing 32, 80
  - shared functionality 30, 40
- DFX-botf.scm 164
- DFXdefault.scm 22
- DFX-depth-phase-snr.scm 218
- DFX-detection.par 80, 290
- DFX-detection.scm 80
- DFX-evch-hydro.par 323
- DFX-evch-hydro.scm 323
- DFX-evch.scm 249
- DFX-hydro-detection.scm 290
- DFX-hydro-recall.scm 312
- DFX-infra-detection.scm 340
- DFX-int-recall.scm 183
- DFX-noiseamp.scm 232
- DFX-originbeam.par 148
- DFX-originbeam.scm 148
- DFX-recall.scm 202
- DPSNR 218
  - architecture 220
  - database attributes written 447
  - data produced 223
  - Estimate Depth-phase SNR* 225, 226
    - Acquire Waveforms* 227
    - architecture 227
    - Find Best Pair* 230
    - Make Amplitude Pairs* 229
    - Make Amplitude Record* 230
    - Measure Amplitudes* 228
    - Write to Database* 230
  - Initialize Site* 224
  - input parameters 221
  - Query for Depth-phase Arrivals* 223

## E

- error handling 76
- extract-gob-attr 111
- extract-gobj-attr 73

## F

- find-det-best-beam 216
- find-detections 119, 307
- find-missing-data 427
- find-recall-best-beam 196
- find-triggers 119, 306
- functional description 30

## G

- GA 6
- gc-wfdata-con 247
- generic objects 21
  - defining 71
- get-data-interval 65
  - functional interface 66
- get-infra-waveforms 349
- get par 75
- get-phases-from-ti-and-beam-
  - recs 174
- get-processing-interval 65
  - functional interface 66
- getspar 75
- global libraries 26
- global Scheme functions 22

## H

- hydro\_features** 29, 449, 450
- hydroacoustic processing 34, 290
- HydroDet* 290
  - Acquire Waveforms* 296
  - architecture 292

▼ Index

- database attributes written 449
- data produced 295
- initialize sites* 295
- input parameters 293
- Make Detections* 297, 304
  - architecture 304
  - Compute SNR* 306
  - extract Parameters* 305
  - Filter Data* 305
  - Generate Detections* 307
  - Generate Triggers* 306
- Measure Hydroacoustic Detection*
  - Features* 300, 307
  - architecture 308
  - Correct Waveform for Instrument Response* 309
  - Estimate Hydroacoustic Features* 309
  - Extract Parameters* 309
  - Revise Detection Attributes* 310
- Revise Onset Times* 298
- Screen and Submit Results to Database* 301
- Screen Detections by Time* 299
- HydroEvch* 323
  - architecture 325
  - database attributes written 450
  - data produced 328
  - Estimate Event Characteristics* 329, 330
    - Acquire Waveforms* 334
    - architecture 332
    - Compute Time Interval* 333
    - Estimate Amplitudes* 335
    - Predict Arrivals* 334
    - Screen and Submit Results* 337
- Initialize Sites* 329
- input parameters 326
- Query for Origins* 328
- 
- I**
- 
- infra\_features* 29, 451
- 
- infra-beam-from-score-boards* 357, 377
- infra-cross-correlate-pairs* 353, 369
- InfraDet* 340
  - Acquire Waveforms* 349
  - architecture 343
  - database attributes written 451
  - data produced 346
  - Estimate Signal Amplitude* 363, 391
    - architecture 392
    - Create Beam-recipe Container* 394
    - Estimate Amplitude* 397
    - Extract Parameters* 393
    - Select Best Measurement* 398
  - Generate Coherence Traces as Beams* 356, 376
    - architecture 377
    - Generate Coherence Trace from Score Boards* 377
    - Generate dbwfdisc Object* 379
    - Write wfdisc Record* 380
  - Generate Coincidence Detection Space (CDS)* 358
  - Generate Primary Detection Space (PDS)* 352, 366
    - architecture 367
    - Cross-correlate Waveforms* 369
    - Evaluate and Refine Slowness Planes* 373
    - Filter Waveforms* 368
    - Generate Slowness Planes* 370
  - Initialize Parameters* 347
  - input parameters 344
  - Remove Redundant Detections* 363, 381
    - architecture 382
    - Create dbcontainers* 383
    - Create dbobjects* 384
    - Delete Redundant Detections* 386
    - Extract Parameters* 383
    - Put dbobjects into dbcontainers* 385
    - Screen for Existing Entities* 387
    - Set arids into dbobjects* 385

- Write Non-Redundant Detections* 366, 399
    - Roll Back Database* 403
    - Test for Empty Input Container* 400
    - Test for Unsuccessful Write* 402
    - Write Container* 402
  - infra-fir-filter-data* 353, 368
  - infra-generate-slowness-planes-cc* 353, 370
  - infra-peak-eval-slow-planes* 353, 373
  - infrasonic processing* 36, 339, 340
  - infra-wvfrm-eval-slow-planes* 358
  - init-<name>-recipe* 348
  - init-<recipe\_name>-recipe* 21
  - initialize-for-residual-update* 210
  - initialize-for-travel-time* 189
  - initialize-net-sites* 155, 238, 261, 329, 412
  - initialize-sites* 173
  - initialize-station-sites* 169
  - initsite* object 46, 72
    - creating 52
    - database objects stored in 47
    - parameters stored in 47
    - recipes stored in 48
    - waveform objects stored in 49
  - instrument** 29
  - instrument-response-files-exist?* 363
  - Interactive Beam on the Fly* 6, 164
    - description 10
  - Interactive Hydroacoustic Recall* 6
    - description 10
    - IntHydroRcl* 312
  - Interactive Seismic Recall* 6, 183
    - description 10
  - interface*
    - external user 38
    - operator 38
    - other IDC systems 37
  - IntHydroRcl*
    - architecture 314
    - database attributes written 450
    - Data Produced 316
    - Extract Station List* 317
    - Initialize Site* 317
    - input parameters 315
    - Perform Recall Signal Processing* 318, 319
      - Acquire Waveforms* 321
      - architecture 320
      - Correct Waveform for Instrument Response* 321
      - Estimate Hydroacoustic Features* 321
      - Generate Detection Object* 320
      - Query Database for Arrivals* 317
  - intrinsic Scheme functions* 22
  - IntSeisRcl* 183
    - architecture 185
    - database attributes written 446
    - data written 188
    - Initialize Parameters* 189
    - Initialize Site* 190
    - Initialize Travel-time Tables* 189, 192
      - architecture 193
      - Query for Sites* 193
      - Read Travel-time Tables* 193
    - input parameters 186
    - Perform Recall Signal Processing* 191, 194
      - Acquire Waveforms* 196
      - architecture 195
      - Estimate Arrival Features and Write to Database* 200
      - Identify Best Beam* 196
      - Identify Depth-phase Arrivals* 195
      - Revise Detection Attributes* 198
  - IPC 27
- ## L
- lastid** 29, 46
  - libgdi* 16, 71

## ▼ Index

*libgobj* 21, 71  
*libloc* 18  
    interface 64  
*libpar* 16, 75  
local Scheme functions 23

**M**

*make-and-write-tm-beams* 93  
*make-beamform-fk* 129  
*make-beam-from-data* 104, 116, 305  
*make-det-amplitudes* 138  
*make-det-beams* 112, 140  
*make-detections* 96, 113  
*make-fk* 128  
*make-gobj-name* 71  
*make-hydro-detections* 297  
*make-null-beam* 104  
*make-pre-existing-arrival-*  
    *beams* 81, 94, 108  
*make-smult* 283  
*man* pages iii  
*Map* 12  
*map-container* 74  
*measure-amplitude* 397  
*measure-arrival-amplitudes* 274,  
    275  
*measure-data-availability* 430  
*measure-detection-amplitude* 228  
*measure-detection-features* 99, 132  
*measure-detection-*  
    *polarization* 131  
*measure-hydro* 309  
*measure-hydro-detection-*  
    *features* 300  
*measure-hydro-recall-*  
    *features* 321  
*measure-infra-amplitude* 364  
*measure-origin-amplitudes* 245,  
    272, 334  
*measure-recall-features* 200  
memory management 77

*mstpar* 75  
*mstspar* 75

**N**

*NoiseAmp* 232  
    architecture 234  
    database attributes written 448  
    data produced 237  
    *Estimate Noise Amplitude* 239, 240  
        *Acquire Waveforms* 244  
        architecture 241  
        *Check Event-to-Station*  
            *Distance* 243  
        *Estimate Origin Amplitudes for Pre-*  
            *dicted Arrivals* 245  
        *Extract Associated Arrivals for this*  
            *Origin* 244  
        *Extract Parameters* 242  
        *Write Estimates for Predicted*  
            *Arrivals* 247  
    *Extract Parameters* 239  
    *Initialize Parameters* 237  
    *Initialize Sites* 238  
    input parameters 235  
    *Query the Database* 238

**O**

*origerr* 29  
*origin* 29  
*OriginBeam*  
    architecture 150  
    database attributes written 445  
    data produced 153  
    *Generate and Write Origin Beams* 155,  
        156  
        *Acquire Waveforms* 159  
        *Extract Parameters* 158  
        *Generate Beams* 161  
        *Identify ti-recipe* 159  
        *Write Beams* 162



*initialize sites* 155  
*input parameters* 151  
*Query for Origins* 154

## P

*parameters* 20  
*par files* 24  
**parrival** 29, 448, 450  
*process-depth-phase-arrival* 225  
*process-origin* 414  
*process-origin-for-initsite* 155, 239, 264, 329  
*process-origins* 413  
*process-qcstats* 426  
*process-recall-arrival* 191, 203, 212, 318

## Q

**QCStats** 423  
     *architecture* 424  
     *Calculate Waveforms Availability* 429, 437  
     *architecture* 438  
     *Compare Requested and Adjusted Available Data Duration* 440  
     *Extract Parameters* 439  
     *Fill-in qc-stats Object with Data Availability* 440  
     *database attributes written* 452  
     *data produced* 426  
     *Extract Quality-control Parameters* 428  
     *Fill-in qc-stats Object* 430  
     *Identify Missing Waveforms* 432  
         *architecture* 433  
         *Seek Match to sta-i and chan-i in Data Container* 435  
         *Set Attributes into qc-stats Object* 435

*Write Results to Database* 436  
     *input parameters* 425  
     *Write Results to Database* 431  
**qcstats** 29, 452  
*qc-waveforms*  
     *functional interface* 70  
*qc-waveforms-all*  
     *functional interface* 70  
*Quality Control Statistics* 423  
     *description* 10  
*query-for-dbarrival*  
     *container* 208  
*query-for-dbarrival-depth-phases* 223  
*query-for-dbarrivals-with-beams* 110  
*query-for-dborigin-by-time* 154, 260  
*query-for-dborigin-container* 412  
*query-for-dbsite-by-sta* 193  
*query-for-lastid* 46  
*query-for-stations-site-container* 173  
*query-for-wfdata-by-sta-chan-time*  
     *functional interface* 58  
*query-for-wfdata-container* 176  
*query-for-wfmem-by-sta-chan-time*  
     *functional interface* 62  
*query-for-wfmem-container* 176

## R

*read-default-travel-time-tables* 64, 174  
*read-detection-waveforms* 81, 91  
*read-hydro-detection-waveforms* 296  
*read-waveforms* 65, 419  
     *functional interface* 69  
*read-waveforms-for-initsite* 159, 244, 270, 334

## ▼ Index

read-waveforms-for-recall 196,214  
 REB 8  
 revise-detection-estimates 97  
 revise-det-onset 125  
 revise-hydro-det-attributes 310  
 revise-hydro-detection-estimates 298  
 revise-recall-estimates 198,216

## S

say-<name>-object-class 71  
 say-beam-object-class 72  
 say-beamrec-object-class 72  
 say-db<table>-object-class 72  
 say-dbconnection-object-class 72  
 say-det-object-class 72  
 say-initsite-object-class 72  
 say-trig-object-class 72  
 say-wfdata-object-class 72  
 say-wfmem-object-class 72  
 Scheme  
   configuration 24  
 Scheme files  
   general structure 24  
 screen-arrival-dbamplitude-by-existing 274,276  
 screen-ceppks-by-existing-ceppks 284  
 screen-complexity-by-existing-complexity 286  
 screen-detection-container 98  
 screen-detections-by-time 299  
 screen-existing-hevch-parrivals 337  
 screen-hydro-detections 301  
 screen-infra-dbarrivals-by-existing 387,389  
 screen-infra-dbdetects-by-existing 387,389  
 screen-infra-dbinfra\_features-by-existing 387,389,390

screen-origin-dbamplitude-by-existing-dbamplitude 273  
 screen-recall-detection 216  
 screen-splp-by-existing-splp 282  
 screen-spvar-by-existing-spvar 284  
 screen-tf-by-existing-tf 278  
 screen-tmf-by-existing-tmf 280  
 SEB 7  
 SegArch 406  
   architecture 408  
   database attributes written 451  
   data produced 411  
   *Initialize Site-specific Parameters* 412  
   input parameters 409  
   *Perform Segment Archiving* 413,415  
     *Acquire Waveforms* 419  
     architecture 416  
     *Calculate Distance-based Time Intervals* 417  
     *Calculate Station-to-Event Distance* 418  
     *Check Waveform Quality* 420  
     *Extract Parameters* 417  
     *Generate Beams* 420  
     *Generate Distance-based Beam Recipes* 419  
     *Write Beams to Database* 422  
     *Query for Origins (in Interval)* 412  
 Segment Archiving 8,406  
   description 10  
 SeisDet 80  
   *Acquire Waveforms* 91  
   architecture 82  
   database attributes written 444  
   data produced 87  
   *Generate and Write TM Beams* 93,101  
     architecture 102  
     *Extract Parameters and Save and Reset stav-len* 103  
     *Generate Beam* 104  
     *Generate Null Beam* 104  
     *Generate STA and LTA* 105  
     *Write TM Beam to UNIX*

- Filesystem* 106
- Generate Pre-existing Arrival*
  - Beams* 94, 108
  - architecture* 109
  - Extract Parameters from Dbarrival*
    - Object (and Set into Detection Object)* 111
  - Generate Detection Beam* 112
  - Generate Detection Object* 111
  - Query Database (Arrivals with Beams)* 110
- Initialize Sites* 88
- input parameters* 84
- Perform Detection Processing* 96, 113
  - architecture* 114
  - Evaluate Triggers and Generate Detections* 119
  - Extract Beam-Name* 115
  - Extract Parameters* 114
  - Generate Detection Beam* 116
  - Generate STA and LTA (and SNR and state)* 116
  - Generate Triggers* 118
- Perform Feature Extraction* 99, 132
  - architecture* 133
  - Create and Write 3-C Amplitudes* 145
  - Create and Write Dbarrival Object* 135
  - Create and Write Dbdetection Object* 142
  - Estimate and Write Detection Amplitude* 138
  - Estimate and Write sbsnr Amplitude* 144
  - Extract Parameters* 134
  - Generate and Write Dbapma Object* 137
  - Generate and Write Detection Beams* 140
- Revise Extracted Estimates* 97, 122
  - architecture* 123
  - Estimate Amplitude (Best Beam)* 127
  - Estimate Polarization* 131
  - Extract Parameters* 124
  - Generate F-k Spectrum* 128
  - Generate F-k Spectrum via Beam* 129
  - Revise Onset Time Estimate* 125
- Screen Defective and Redundant Detections* 98
- SeisEvch* 249
  - architecture* 251
  - database attributes written* 448
  - data produced* 259
  - Estimate Event Characteristics* 264, 267
    - Acquire Waveforms* 270
    - architecture* 267
    - Extract Parameters* 270
    - Perform Complexity Processing* 285
    - Perform First Motion Amplitude Processing* 275
    - Perform Observed Amplitude Processing* 273
    - Perform Short-period-versus-long-period Processing* 281
    - Perform Source Multiplicity Processing* 283
    - Perform Theoretical Amplitude Processing* 271
    - Perform Third-moment-of-frequency Processing* 279
    - Perform Time-frequency Processing* 277
  - Initialize Sites* 261
  - input parameters* 252
  - Query for Origins* 259
- seismic processing* 32, 80
- select-dbamplitude-object-to-report* 398
- sensor** 29
- set-cvar!** 75
- set-cvar-from-file!** 47, 75
- set-gobj-attr** 111
- set-gobj-attr!** 73
- SigPro** 12
- site** 29

## ▼ Index

**sitechan** 29  
**site-hydro-blocked** 333  
**site par file** 25  
**software configuration hierarchy** 3  
**splp** 29, 448  
**spvar** 29, 449  
*StaPro* 6  
**station processing** 65  
**submit-aux** 162, 247  
**submit-container-db** 44, 139, 146,  
182, 380, 402  
**submit-db-hydro-results** 302  
**submit-object-db** 44, 136, 138, 143

## T

**thirdmom** 29, 449  
**throw-tag-error** 76  
**timefreq** 30, 449  
*trigger* object 72  
*tuxshell* 27, 37

## U

**update-or-insert-<type>-by-  
<field>** 45

## W

*WaveExpert* 6  
**waveforms**  
    **reading** 58  
**wfdata** object 58, 72  
    **contents** 59  
**wfdisc** 30, 445, 446, 451, 452  
**wfmem** object 61, 72  
**wftag** 30, 445, 452  
**write-db-beam** 379  
**write-qcstats** 431, 436  
**write-tm-sta** 106